O'REILLY®

Fifth Edition

Also covers .NET MAUI & Unity

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET

Andrew Stellman
& Jennifer Greene

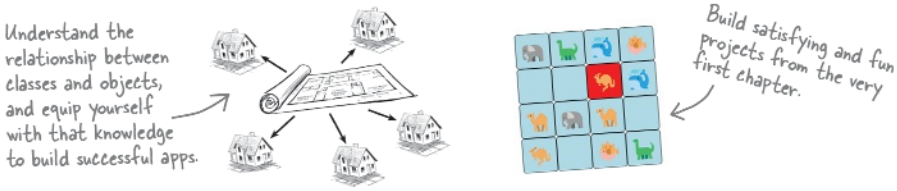
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Create apps, games, and more using this engaging, highly visual introduction to C#, .NET, and software development. You'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, completing hands-on exercises, and building real-world applications. Interested in a development career? You'll learn important development techniques and ideas—just like many others who've learned to code with this book and are now professional developers, team leads, coding streamers, and more. There's no experience required except the desire to learn. And this is the best place to start.



What's so special about this book?

If you've read a Head First book, you know what to expect: a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With this book, you'll learn C# through a multisensory experience that engages your mind—rather than a text-heavy approach that puts you to sleep.

"Thank you so much! Your books have helped me to launch my career."

—Ryan White
Game Developer

"In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!"

—Gerald Versluis
Senior Software Engineer at Microsoft

"Andrew and Jennifer have written a concise, authoritative, and, most of all, fun introduction to C# development."

—Jon Galloway
Senior Program Manager on the .NET Community Team at Microsoft

C# / .NET

US $79.99 **CAN $99.99**

ISBN: 978-1-098-14178-3

O'REILLY®

Praise for Head First C#

"In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!"

—Gerald Versluis, Senior Software Engineer at Microsoft

"*Head First C#* started my career as a software engineer and backend developer. I am now leading a team in a tech company and an open source contributor."

—Zakaria Soleymani, Development Team Lead

"Thank you so much! Your books have helped me to launch my career."

—Ryan White, Game Developer

"If you're a new C# developer (welcome to the party!), I highly recommend *Head First C#*. Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development. I wish I'd had this book when I was first learning C#!"

—Jon Galloway, Senior Program Manager on the .NET Community Team, Microsoft

"Not only does *Head First C#* cover all the nuances it took me a long time to understand, it has that Head First magic going on where it is just a super fun read."

—Jeff Counts, Senior C# Developer

"*Head First C#* is a great book with fun examples that keep learning interesting."

—Lindsey Bieda, Lead Software Engineer

"*Head First C#* is a great book, both for brand-new developers and developers like myself coming from a Java background. No assumptions are made as to the reader's proficiency, yet the material builds up quickly enough for those who are not complete newbies—a hard balance to strike. This book got me up to speed in no time for my first large-scale C# development project at work—I highly recommend it."

—Shalewa Odusanya, Principal

"*Head First C#* is an excellent, simple, and fun way of learning C#. It's the best piece for C# beginners I've ever seen—the samples are clear, the topics are concise and well written. The mini-games that guide you through the different programming challenges will definitely stick the knowledge to your brain. A great learn-by-doing book!"

—Johnny Halife, Partner

"*Head First C#* is a comprehensive guide to learning C# that reads like a conversation with a friend. The many coding challenges keep it fun, even when the concepts are tough."

—Rebeca Dunn-Krahn, Founding Partner, Sempahore Solutions

Praise for Head First C#

"I've never read a computer book cover to cover, but this one held my interest from the first page to the last. If you want to learn C# in depth and have fun doing it, this is THE book for you."

—Andy Parker, fledgling C# Programmer

"It's hard to really learn a programming language without good, engaging examples, and this book is full of them! *Head First C#* will guide beginners of all sorts to a long and productive relationship with C# and the .NET Framework."

—Chris Burrows, Software Engineer

"With *Head First C#*, Andrew and Jenny have presented an excellent tutorial on learning C#. It is very approachable while covering a great amount of detail in a unique style. If you've been turned off by more conventional books on C#, you'll love this one."

—Jay Hilyard, Director and Software Security Architect, and author of *C# 6.0 Cookbook*

"I'd recommend this book to anyone looking for a great introduction into the world of programming and C#. From the first page onward, the authors walk the reader through some of the more challenging concepts of C# in a simple, easy-to-follow way. At the end of some of the larger projects/labs, the reader can look back at their programs and stand in awe of what they've accomplished."

—David Sterling, Principal Software Developer

"*Head First C#* is a highly enjoyable tutorial, full of memorable examples and entertaining exercises. Its lively style is sure to captivate readers—from the humorously annotated examples to the Fireside Chats, where the abstract class and interface butt heads in a heated argument! For anyone new to programming, there's no better way to dive in."

—Joseph Albahari, inventor of LINQPad, and coauthor of *C# 12 in a Nutshell* and *C# 12 Pocket Reference*

"[*Head First C#*] was an easy book to read and understand. I will recommend this book to any developer wanting to jump into the C# waters. I will recommend it to the advanced developer that wants to understand better what is happening with their code. [I will recommend it to developers who] want to find a better way to explain how C# works to their less-seasoned developer friends."

—Giuseppe Turitto, Director of Engineering

"Andrew and Jenny have crafted another stimulating Head First learning experience. Grab a pencil, a computer, and enjoy the ride as you engage your left brain, right brain, and funny bone."

—Bill Mietelski, Advanced Systems Analyst

"Going through this *Head First C#* book was a great experience. I have not come across a book series which actually teaches you so well.…This is a book I would definitely recommend to people wanting to learn C#."

—Krishna Pala, MCP

Praise for the Head First Approach

"I received the book yesterday and started to read it…and I couldn't stop. This is definitely très 'cool.' It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma, IBM Distinguished Engineer, and coauthor of *Design Patterns*

"One of the funniest and smartest books on software design I've ever read."

— Aaron LaBerge, SVP Technology & Product Development, ESPN

"What used to be a long trial and error learning process has now been reduced neatly into an engaging paperback."

— Mike Davidson, former VP of Design, Twitter, and founder of Newsvine

"Elegant design is at the core of every chapter here, each concept conveyed with equal doses of pragmatism and wit."

— Ken Goldstein, Executive VP & Managing Director, Disney Online

"Usually when reading through a book or article on design patterns, I'd have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.

"While other books on design patterns are saying 'Bueller…Bueller…Bueller…' this book is on the float belting out 'Shake it up, baby!'"

— Eric Wuehler

"I literally love this book. In fact, I kissed this book in front of my wife."

— Satish Kumar

Related books from O'Reilly

We love this book so much!

C# 12 in a Nutshell *by Joseph Albahari*

Unity Game Development Cookbook, 2nd Edition
by Paris Buttfield-Addison, Jon Manning, and Tim Nugent

Programming C# 12 *by Ian Griffiths*

Functional Programming with C# *by Simon J. Painter*

C# 12 Pocket Reference *by Joseph Albahari and Ben Albahari*

Learning Blazor *by David Pine*

Other books in O'Reilly's *Head First* series

Head First Android Development

Head First Design Patterns

Head First Git

Head First Python

Head First Go

Head First HTML and CSS

Head First iPhone and iPad Development

Head First Java

Head First JavaScript Programming

Head First Learn to Code

Head First Object-Oriented Analysis and Design

Head First Programming

Head First Python

Head First Software Architecture

Head First SQL

Head First Swift

Head First Web Design

Head First C#

Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First C#

by Andrew Stellman and Jennifer Greene

Copyright © 2024 Andrew Stellman and Jennifer Greene. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra and Bert Bates

Series Advisors: Eric Freeman and Elisabeth Robson

Acquisitions Editor: Brian Guerin

Development Editor: Michele Cronin

Production Editor: Katherine Tozer

Proofreader: Piper Editorial Consulting, LLC

Indexer: Potomac Indexing, LLC

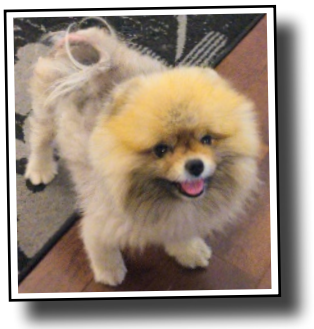
Cover Design: Susan Thompson, based on a series design by Ellie Volckhausen

Cover and Interior Illustrations: José Marzan Jr.

Page Viewers: Samosa the Pomeranian and Rosalie the Australian Labradoodle

Printing History:

November 2007: First Edition
May 2010: Second Edition
August 2013: Third Edition
December 2020: Fourth Edition
July 2024: Fifth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

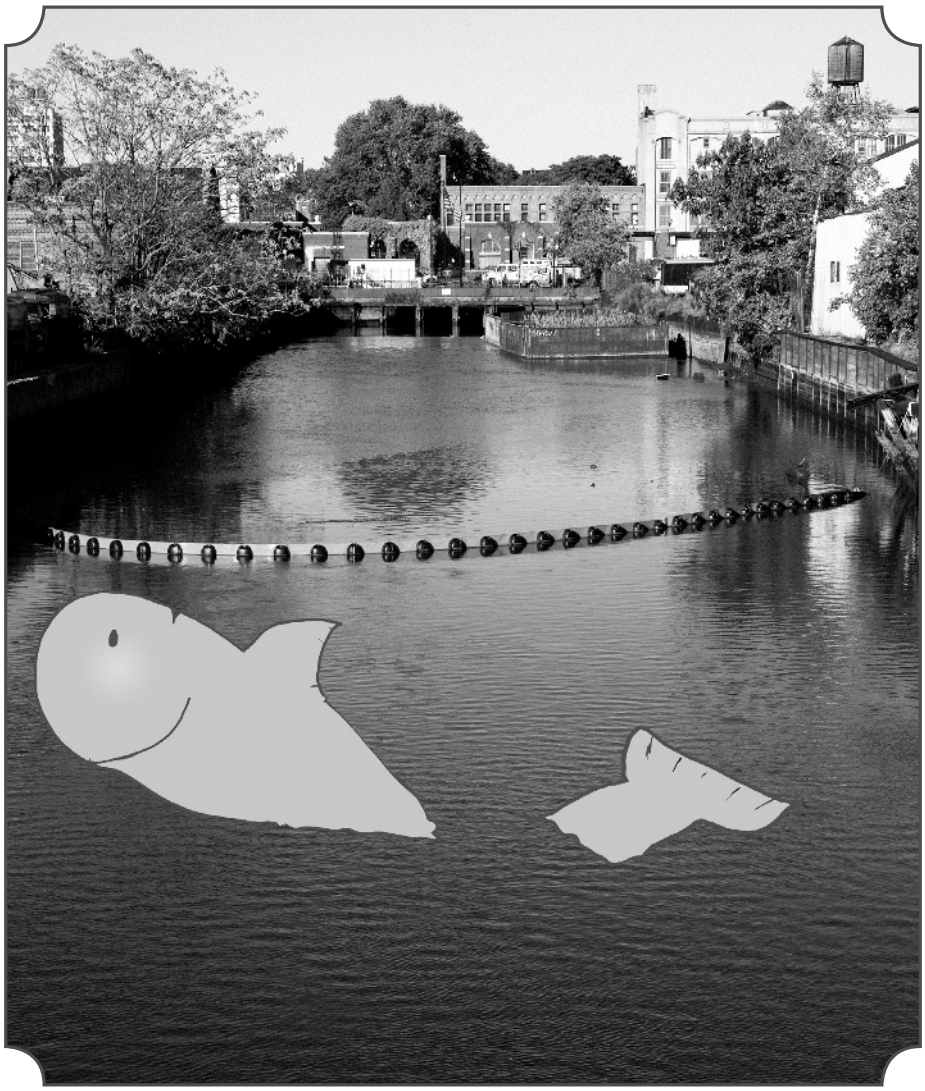
While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No pigeons, ducks, ostriches, elephants, dogs, hippos, chameleons, bees, clowns, aliens, game masters, jewel thieves, or superheroes were harmed in the making of this book.

ISBN: 978-1-098-14178-3

[LSI] [2024-07-08]

This book is dedicated to the loving memory of Sludgie the Whale,
who swam to Brooklyn on April 17, 2007.



You were only in our canal for a day,
but you'll be in our hearts forever.

Thanks for reading our book! We really love writing about this stuff, and we hope you get a lot out of it...

...because we know you're going to have a great time learning C#.

Andrew

Jenny

This photo (and the photo of the Gowanus Canal) by Nisha Sondhe

Andrew Stellman, despite being raised a New Yorker, has lived in Minneapolis, Geneva, and Pittsburgh…*twice*, first in the early '90s when he graduated from Carnegie Mellon's School of Computer Science, and then again in the early 2000s while he and Jenny were starting their consulting business and writing their first book for O'Reilly.

Andrew's first job after college was building software at a record company, EMI-Capitol Records—which actually made sense, as he went to LaGuardia High School of Music & Art and the Performing Arts to study cello and jazz bass guitar. He and Jenny first worked together at a company on Wall Street that built financial software, where he was managing a team of programmers. Over the years he's architected large-scale real-time backend systems, managed large international software teams, been a senior vice president at a major financial firm, and consulted for companies, schools, and organizations, including Microsoft, the National Bureau of Economic Research, and MIT. He's had the privilege of working with some pretty amazing developers during that time, and likes to think that he's learned a few things from them.

When he's not writing books, Andrew keeps himself busy writing useless (but fun) software, playing (and making) both music and video games, practicing krav maga, tai chi, and aikido, and owning a crazy Pomeranian.

Jennifer Greene studied philosophy in college but, like everyone else in the field, couldn't find a job doing it. Luckily, she's a great software engineer, so she started out working at an online service, and that's the first time she really got a good sense of what good software development looked like.

She moved to New York in 1998 to work on software quality at a financial software company. She's managed teams of developers, testers, and PMs on software projects in media and finance since then.

Jenny has traveled all over the world to work with different software teams and build all kinds of cool projects.

She loves traveling, watching Bollywood movies, reading the occasional comic book, playing video games, and hanging out with her Australian Labradoodle.

Jenny and Andrew have been building software and writing about software engineering together since they first met in 1998. Their first book, **Applied Software Project Management**, was published by O'Reilly in 2005. Other Stellman and Greene books published by O'Reilly include **Beautiful Teams** (2009), **Learning Agile** (2014), **Head First Agile** (2017), and their first book in the Head First series, **Head First PMP** (2007), now in its fourth edition.

They founded Stellman & Greene Consulting in 2003 to build a really fascinating software project for scientists studying herbicide exposure in Vietnam veterans. In addition to building software and writing books, they've consulted for companies and spoken at conferences and meetings of software engineers, architects, and project managers.

Learn more about them on their website, **Building Better Software (https://www.stellman-greene.com)**.

☮❤️🐙 Jenny and Andrew

Table of Contents (Summary)

	Intro	xxix
1	Start building apps with C#: *Build something great...fast!*	1
2	Variables, statements, and methods: *Dive into C# code*	65
	Unity Lab 1: Explore C# with Unity	111
3	Namespaces and classes: *Organizing your code*	127
4	Data, types, objects, and references: *Managing your app's data*	189
	Unity Lab 2: Write C# Code for Unity	257
5	Encapsulation: *How objects keep their secrets*	271
6	Inheritance: *Your object's family tree*	325
	Unity Lab 3: GameObject Instances	403
7	Interfaces, casting, and is: *Making classes keep their promises*	415
8	Enums and collections: *Organizing your data*	473
	Unity Lab 4: User Interfaces	539
9	LINQ and lambdas: *Get control of your data*	553
10	Reading and writing files: *Save the last byte for me*	621
	Unity Lab 5: Raycasting	673
11	Captain Amazing: *The Death of the Object*	687
12	Exception handling: *Putting out fires gets old*	731
	Unity Lab 6: Scene Navigation	763

Table of Contents (the real thing)

Intro

Your brain on C#. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing C#?

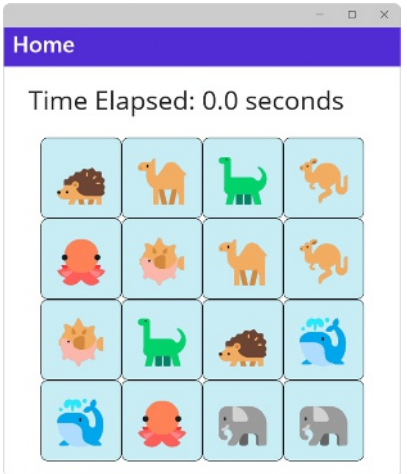
Who is this book for?	xxx
We know what you're thinking	xxxi
Metacognition	xxxiii
Bend your brain into submission	xxxv
Read me	xxxvi
The technical review team	xxxviii
Acknowledgments	xl

1 start building apps with C#

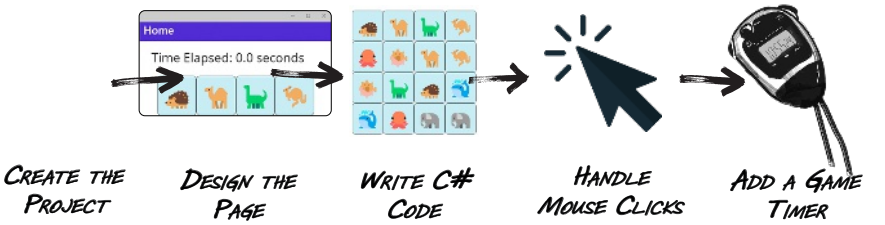
Build something great...fast!

Want to build great apps…right now?

With C#, you've got a modern programming language and a **valuable tool** at your fingertips. And with **Visual Studio**, you've got an amazing development environment with highly intuitive features that make coding as easy as possible. Not only is Visual Studio a great tool for writing code, it's also a **really effective learning tool** for exploring C#. Sound appealing? ***Let's get coding!***



Learn C#…and learn to become a great developer	2
Write code and explore C# with Visual Studio	3
Install Visual Studio Community Edition	4
Run Visual Studio	5
Create and run your first C# project in Visual Studio	6
You can use Visual Studio Code with *Head First C#*	12
Create and run your first project in Visual Studio Code	14
Set up Visual Studio Code for the next project	17
Let's build a game!	18
Create a .NET MAUI project in Visual Studio	22
Run your new .NET MAUI app	24
MAUI apps work on all of your devices	25
Start editing your XAML code	27
Use a FlexLayout to make a grid of animal buttons	34
Write C# code to add the animals to the buttons	38
Run your app!	46
Visual Studio makes it easy to use Git	51
Add C# code to handle mouse clicks	52
Add a timer to your game's code	60
Finish the code for your game	62



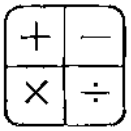
2 variables, statements, and methods

Dive into C# code

You're not just an IDE user. You're a developer.

You can get a lot of work done using the IDE, but there's only so far it can take you. Visual Studio is one of the most advanced software development tools ever made, but a **powerful IDE** is only the beginning. It's time to ***dive* in to C# code**: how it's structured, how it works, and how you can take control of it…because there's no limit to what you can get your apps to do.

Take a closer look at the files in your console app	66
Statements are the building blocks for your apps	68
Statements live inside methods	69
Your methods use variables to work with data	70
Generate a new method to work with variables	72
Add code that uses operators to your method	73
Use the debugger to watch your variables change	74
Use code snippets to help write loops	76
Use operators to work with variables	77
if statements make decisions	78
Loops perform an action over and over	79
Controls drive the mechanics of your user interfaces	88
Other controls you'll use in this book	89
Create a new app to experiment with controls	91
Explore your new MAUI app and figure out how it works	92
Add an Entry control to your app	96
Add properties to your Entry control	97
Make your Entry control update a Label control	98
Combine horizontal and vertical stack layouts	103
Add a Picker control to display a list of choices	104

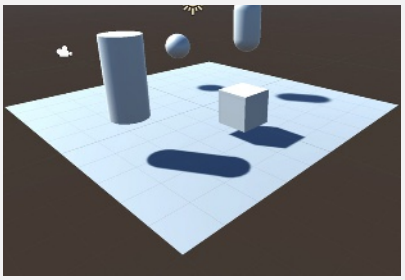


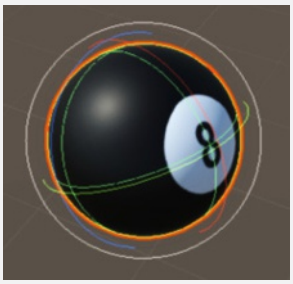
Unity Lab 1

Explore C# with Unity

Welcome to your first **Head First C# Unity Lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**. Unity will be a really valuable tool for that. In this lab, you can begin practicing what you've learned about C# in Chapters 1 and 2.

Unity is a powerful tool for game design	112
Download Unity Hub	113
Use Unity Hub to create a new project	114
Your scene is a 3D environment	116
Unity games are made with GameObjects	117
Use the Move Gizmo to move your GameObjects	118
The Inspector shows your GameObject's components	119
Add a material to your Sphere GameObject	120
Rotate your sphere	123
Get creative!	126





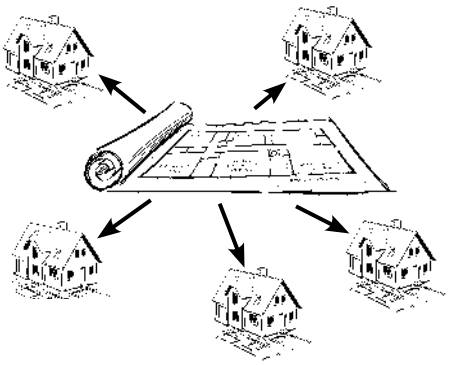
namespaces and classes

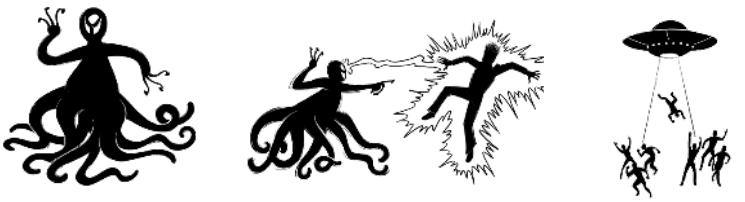
3 Organizing your code

Great developers keep their code and data organized.

What's the first thing you do when you're creating an app? You think about what **it's supposed to do**, whether you're solving a problem, creating a game, or just having fun. But it's not always obvious how individual statements fit into your app's bigger picture…and that's where **classes** come in. They let you **organize your code** around the features you're creating and the problems the app needs to solve. Classes can help you **organize your data** too, by using them to create **objects** that represent any "thing" your app needs to know about—and the classes that you design serve as "blueprints" for the objects used in your app.

Classes help you organize your code	128
Some methods take parameters and return a value	130
Let's build an app that picks random cards	132
Create an app with a Main method	134
Use Quick Actions to remove unnecessary using lines	138
Convert between namespace styles	139
Use the new keyword to create an array of strings	140
Build a paper prototype for a classic game	148
Build a MAUI version of your random card app	150
Reuse your CardPicker class	154
Add a using directive to use code in another namespace	155
You use a class to build an object	159
A better solution for Ana…brought to you by objects	161
An instance uses fields to keep track of things	165
Use intuitive class and method names	172
Build a class to work with some guys	178
Use the C# Interactive window or csi to run C# code	188



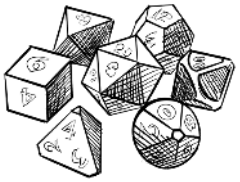


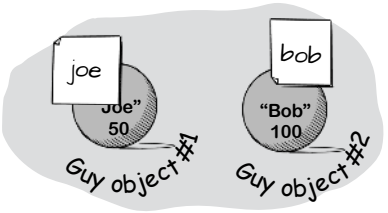
4 data, types, objects, and references

Managing your app's data

Data and objects are the building blocks of your apps.

What would your apps be without data? Think about it for a minute. Without data, your programs are…well, it's actually hard to imagine writing code without data. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types** and **references**, see how to work with data in your program, and even learn a few more things about **objects** (guess what…objects are data, too!).





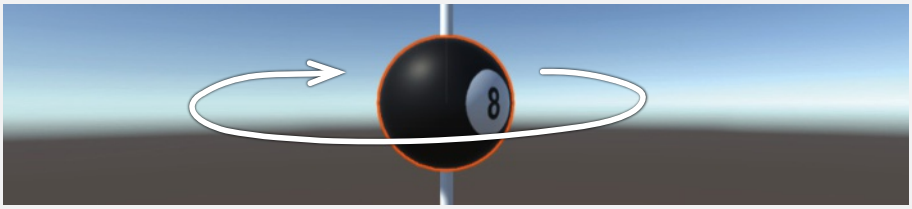
A variable's type determines what kind of data it can store	192
C# has several types for storing integers	193
Let's talk about strings	195
A literal is a value written directly into your code	196
Casting lets you copy values that C# can't automatically convert to another type	202
C# does some conversions automatically	205
Use reference variables to access your objects	222
References are like sticky notes for your objects	223
Multiple references and their side effects	226
Two references mean TWO variables that can change the same object's data	233
Objects use references to talk to each other	234
Arrays hold multiple values	236
null means a reference points to nothing	241
Use the string? type when a string might be null	243
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!	246
Grid controls	248
Create the Sloppy Joe's menu app and set up the grid	250
Use the SetValue method to change a control's semantic properties	256

Unity Lab 2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**. In this lab, you'll get more practice writing C# code for a project in Unity.

C# scripts add behavior to your GameObjects	258
Add a C# script to your GameObject	259
Write C# code to rotate your sphere	260
Add a breakpoint and debug your game	262
Use the debugger to understand Time.deltaTime	263
Add a cylinder to show where the Y axis is	264
Add fields to your class for the rotation angle and speed	265
Use Debug.DrawRay to explore how 3D vectors work	266
Run the game to see the ray in the Scene view	267
Rotate your ball around a point in the scene	268
Use Unity to take a closer look at rotation and vectors	269
Get creative!	270



5 encapsulation

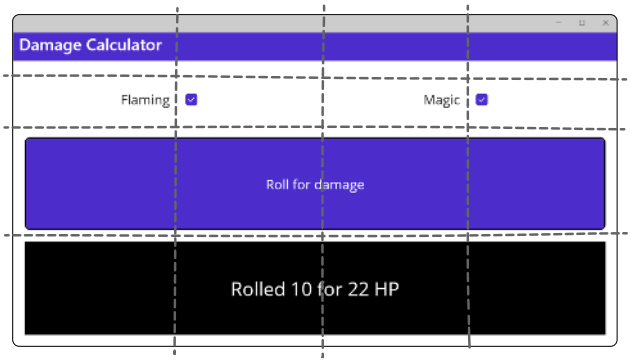
How objects keep their secrets

Ever wished for a little more privacy?

Sometimes your objects feel the same way. Just like you don't want anybody you don't trust reading your journal or paging through your bank statements, good objects don't let ***other*** objects go poking around their fields. In this chapter, you're going to learn about the power of **encapsulation**, a way of programming that helps you make code that's flexible, easy to use, and difficult to misuse. You'll **make your objects' data private**, and add **properties** to protect how that data is accessed—and you'll keep your objects' important data from **leaking out to other objects** so they don't accidentally misuse it.

Let's help Owen roll for damage	272
Create a console app to calculate damage	273
Design a MAUI version of the damage calculator app	275
Use Debug.WriteLine to print diagnostic information	281
Use encapsulation to control access to your class's methods and fields	286
Private fields and methods can only be accessed from instances of the same class	288
Why encapsulation? Think of an object as an opaque box…	293
Let's use encapsulation to improve the SwordDamage class	297
Write a console app to test the PaintballGun class	299
Auto-implemented properties simplify your code	302
Use a private setter to create a read-only property	303
Use a constructor with parameters to initialize properties	305
Specify arguments when you use the new keyword	306
Initialize fields and properties inline or in the constructor	313

- To find the number of hit points (HP) of damage for a sword attack, roll 3d6 (three 6-sided dice) and add "base damage" of 3HP.
- Some swords are flaming, which causes an extra 2HP of damage.
- Some swords are magic. For magic swords, the 3d6 roll is multiplied by 1.75 and rounded down, and the base damage and flaming damage are added to the result.

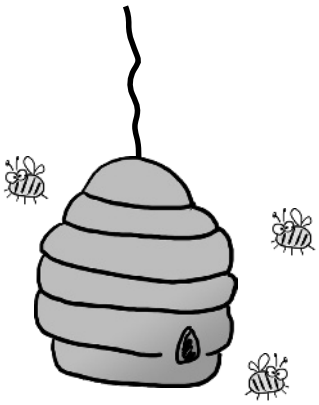


inheritance

6 Your object's family tree

Sometimes you DO want to be just like your parents.

Ever run across a class that ***almost*** does exactly what you want ***your*** class to do? Found yourself thinking that if you could just ***change a few things***, that class would be perfect? With **inheritance**, you can **extend** an existing class so your new class gets all of its behavior—with the **flexibility** to make changes to that behavior so you can tailor it however you want. Inheritance is one of the most powerful concepts and techniques in the C# language: with it, you can **avoid duplicate code**, **model the real world** more closely, and end up with apps that are **easier to maintain** and **less prone to bugs**.



Use a switch statement to match several candidates 327

When your classes use inheritance, you only need to write your code once 330

How would you design a zoo simulator? 332

Any place where you can use a base class, you can use one of its subclasses instead 338

A subclass can override methods to change or replace members it inherited 344

Build an app to explore virtual and override 352

A subclass can hide methods in the base class 354

Use the override and virtual keywords to inherit behavior 356

A class should do one thing 366

Build a Beehive Management System 370

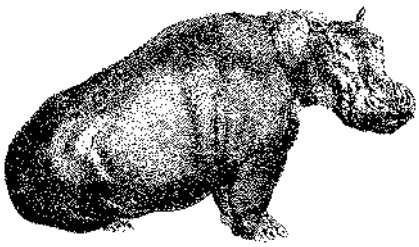
Feedback drives your beehive management game 388

The Beehive Management System is turn-based… now let's convert it to real-time 390

An abstract class is an intentionally incomplete class 394

Abstract properties work just like abstract methods 398

The Deadly Diamond of Death 401

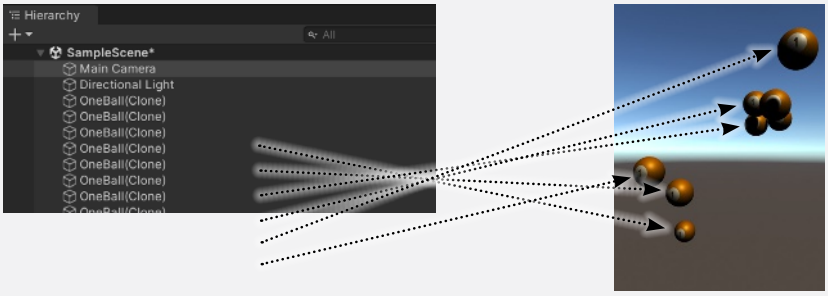


Unity Lab 3

GameObject Instances

C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

Let's build a game in Unity!	404
Create a new material inside the Materials folder	405
Spawn a billiard ball at a random point in the scene	406
Use the debugger to understand Random.value	407
Turn your GameObject into a prefab	408
Create a script to control the game	409
Attach the GameController script to the Main Camera	410
Press Play to run your code	411
Use the Inspector to work with GameObject instances	412
Use physics to keep balls from overlapping	413
Get creative!	414



interfaces, casting, and is

7 Making classes keep their promises

Need an object to do a specific job? Use an interface.

Sometimes you need to group your objects together based on the things they can do rather than the classes they inherit from—and that's where interfaces come in. You can use an interface to define a **specific job**. Any instance of a class that **implements** the interface is *guaranteed to do that job*, no matter what other classes it's related to. To make it all work, any class that implements an interface must promise to fulfill all of its obligations…or the compiler will break its kneecaps, see?

The beehive is under attack! 416

We could use casting to call the DefendHive method… 417

An interface defines methods and properties that a class must implement… 418

Interfaces let unrelated classes do the same job 419

Get a little practice using interfaces 420

You can't instantiate an interface, but you can reference an interface 426

Interface references are ordinary object references 429

The RoboBee 4000 can do a worker bee's job without using valuable honey 430

What if we want different animals to swim or hunt in packs? 438

Use interfaces to work with classes that do the same job 439

Safely navigate your class hierarchy with is 440

C# has another tool for safe type conversion: the as keyword 441

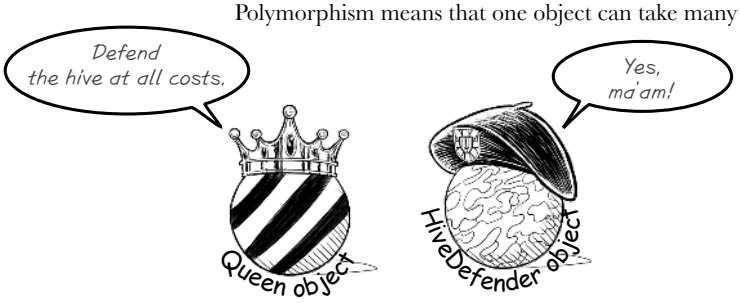
Use upcasting and downcasting to move up and down a class hierarchy 442

Upcasting and downcasting work with interfaces too 446

Default implementations give bodies to interface methods 456

Data binding updates MAUI controls automatically 459

Polymorphism means that one object can take many different forms 469



enums and collections

8 Organizing your data

Data isn't always as neat and tidy as you'd like it to be.

In the real world, you don't receive your data in tidy little bits and pieces. No, your data's going to come at you in loads, piles, and bunches. You'll need some pretty powerful tools to organize all of it—and lucky for us, C# has just the tools we need. **Enums** are types that let you define valid values to categorize your data. **Collections** are special objects that store many values, letting you **store, sort, and manage** all the data that your apps need to pore through. That way, you can spend your time thinking about writing apps to work with your data, and let the collections worry about keeping track of it for you.

The rarely played Duke of Oxen card.

If a constructor just sets fields, use a primary constructor instead 474
A primary constructor can extend a base constructor 475
Enums let you work with a set of valid values 477
Enums let you represent numbers with names 478
Lists make it easy to store collections of…anything 483
Let's build an app to store shoes 487
Generic collections can store any type 490
You can use collection expressions to create Lists 496
IComparable<Duck> helps your List sort its Ducks 499
Create an instance of your comparer object 501
Comparers can do complex comparisons 502
You can upcast an entire list using IEnumerable<T> 510
The Dictionary functionality rundown 513
CollectionView is a MAUI control built for displaying collections 524
ObservableCollection is a collection made for data binding 525
Use XAML to instantiate your objects for data binding 529
Modify your app to use a resource Dictionary 530
Modify the event handlers to use the resource Dictionary 532
Use what you've learned to build an app with two decks 533

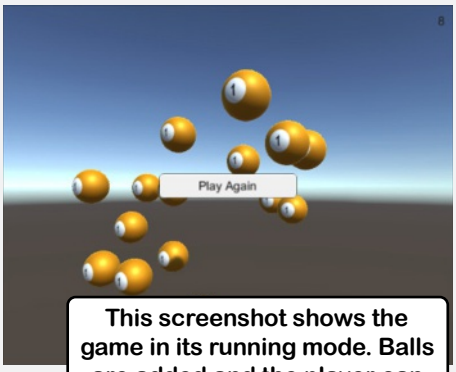
Sorted by kind of duck...

Unity Lab 4

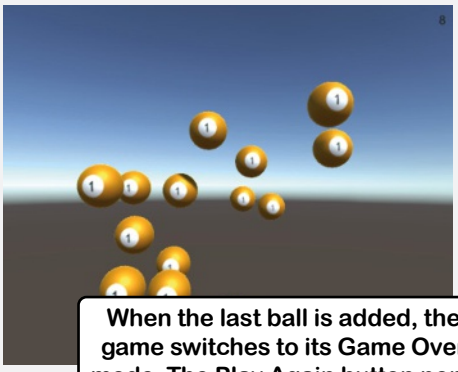
User Interfaces

In the last Unity Lab you started to build a game, using a prefab to create GameObject instances that appear at random points in 3D space and fly in circles. This Unity Lab picks up where the last one left off, allowing you to apply what you've learned about interfaces in C# and more.

Add a score that goes up when the player clicks a ball	540
Add two different modes to your game	541
Add game mode to your game	542
Add a UI to your game	544
Set up the Text that will display the score in the UI	545
Add a button that calls a method to start the game	546
Make the Play Again buton and Score Text work	547
Finish the code for the game	548
Get creative!	552



This screenshot shows the game in its running mode. Balls are added and the player can click on them to score.



When the last ball is added, the game switches to its Game Over mode. The Play Again button pops up and no more balls get added.

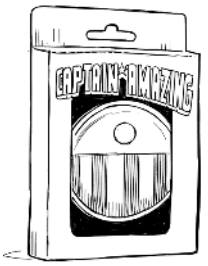
9 LINQ and lambdas

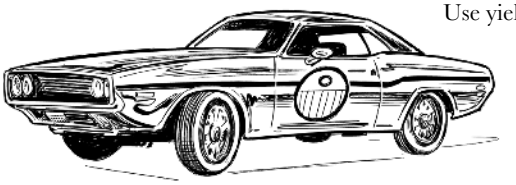
Get control of your data

It's a data-driven world…we all need to know how to live in it.

Gone are the days when you could program for days, even weeks, without dealing with **loads of data**. Today, ***everything is about data***, and that's where **LINQ** comes in. LINQ is a feature of C# and .NET that not only lets you **query data** in your .NET collections in an intuitive way, but lets you **group data** and **merge data from different data sources**. You'll use **anonymous objects** to manage your data in new and interesting ways. You'll add **unit tests** to make sure your code is working the way you want. Once you've got the hang of wrangling your data into manageable chunks, you can use **lambda expressions** to refactor your C# code to make it easier to work with and more expressive.

Jimmy's a Captain Amazing super-fan…	554
Use LINQ to query your collections	556
Use a LINQ query to finish the app for Jimmy	564
The var keyword lets C# figure out variable types for you	566
LINQ is versatile	572
Use a group query to separate your sequence into groups	574
Use join queries to merge data from two sequences	577
Use the new keyword to create anonymous types	578
Unit tests help you make sure your code works	587
Start writing your first test method	588
One project can only access public classes in another project	590
Use the Arrange-Act-Assert pattern to write an effective test	591
Write a unit test for the GetReviews method	594
Use the => operator to create lambda expressions	598
Use the ?: operator to make your lambdas make choices	603
LINQ queries are made up of methods	604
LINQ declarative syntax can be refactored into chained methods	606
Use the => operator to create switch expressions	609
Explore the Enumerable class	613
Use yield return to create your own sequences	615



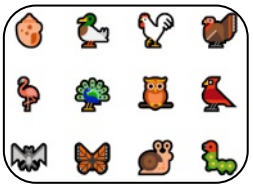


10 reading and writing files

Save the last byte for me

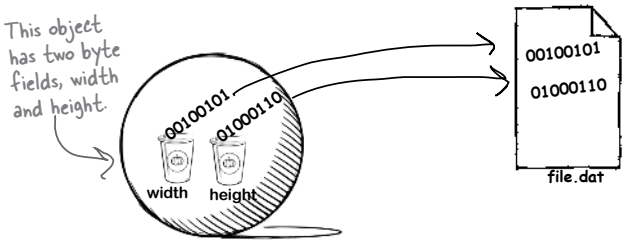
Sometimes it pays to be persistent.

So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**, and then how to **read that information back in** from a file. You'll learn about **streams**, and how to store your objects in files with **serialization**, and get down to the actual bits and bytes of **hexadecimal**, **Unicode**, and **binary data**.



.NET uses streams to read and write data	622
Different streams read and write different things	623
Use a StreamReader to read a file	629
Use the static File and Directory classes to work with files and directories	634
IDisposable makes sure objects are closed properly	637
Avoid filesystem errors with using statements	638
Use a MemoryStream to stream data to memory	639
What happens to an object when it's serialized?	645
Use JsonSerializer to serialize your objects	648
JSON only includes data, not specific C# types	651
C# strings are encoded with Unicode	655
.NET uses Unicode to store characters and text	658
C# can use byte arrays to move data around	660
Use a BinaryWriter to write binary data	661
Use BinaryReader to read the data back in	662
Use StreamReader to build a hex dumper	665
Use Stream.Read to read bytes from a stream	666
Modify your hex dumper to read directly from the stream	667
Run your app from the command line	668


```
0000: 45 6c 65 6d 65 6e 74 61 Elementa
0005: 72 79 2c 20 6d 79 20 64 ry, my d
0010: 65 61 72 20 57 61 74 73 ear Wats
0015: 6f 6e 21                on!
```

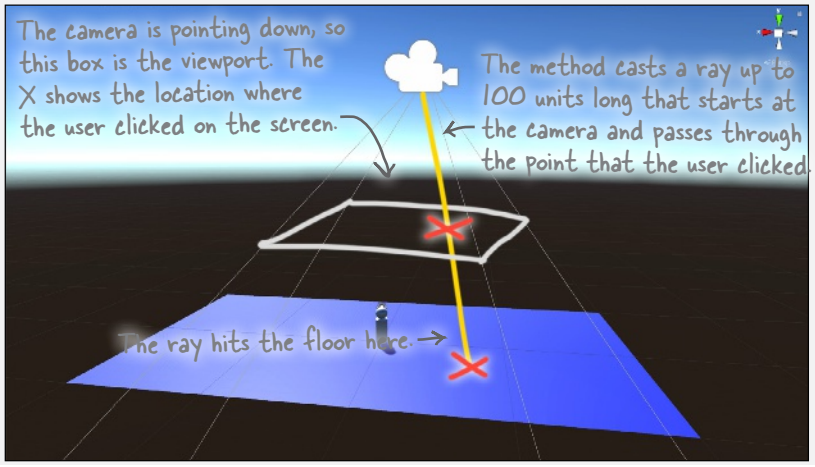


Unity Lab 5

Raycasting

When you set up a scene in Unity, you're creating a virtual 3D world for the characters in your game to move around in. But in most games, things aren't directly controlled by the player. So how do these objects find their way around a scene? In this lab, we'll look at how C# can help.

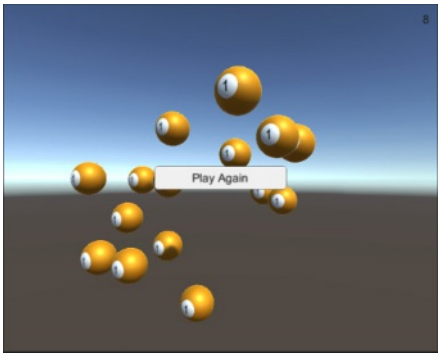
Create a new Unity project and start to set up the scene	674
Set up the camera	675
Create a GameObject for the player	676
Introducing Unity's navigation system	677
Install the AI Navigation package	678
Things you'll do with navigation	679
Set up the NavMesh	680
Make your player automatically navigate the play area	683



CAPTAIN AMAZING

THE DEATH OF THE OBJECT

Head First C#	
Four bucks	Chapter 11



The life and death of an object 690

Use the GC class (with caution) to force garbage collection 691

Your last chance to DO something… your object's finalizer 692

When EXACTLY does a finalizer run? 693

Finalizers can't depend on other objects 695

A struct looks like an object… 699

Values get copied; references get assigned 700

Structs are value types; objects are reference types 701

The stack versus the heap: more on memory 703

Use out parameters to make a method return more than one value 706

Pass by reference using the ref modifier 707

Use optional parameters to set default values 708

A null reference doesn't refer to any object 709

Non-nullable reference types help you avoid NREs 710

Nullable value types can be null…and handled safely 713

The null-coalescing operator ?? checks for nulls automatically 714

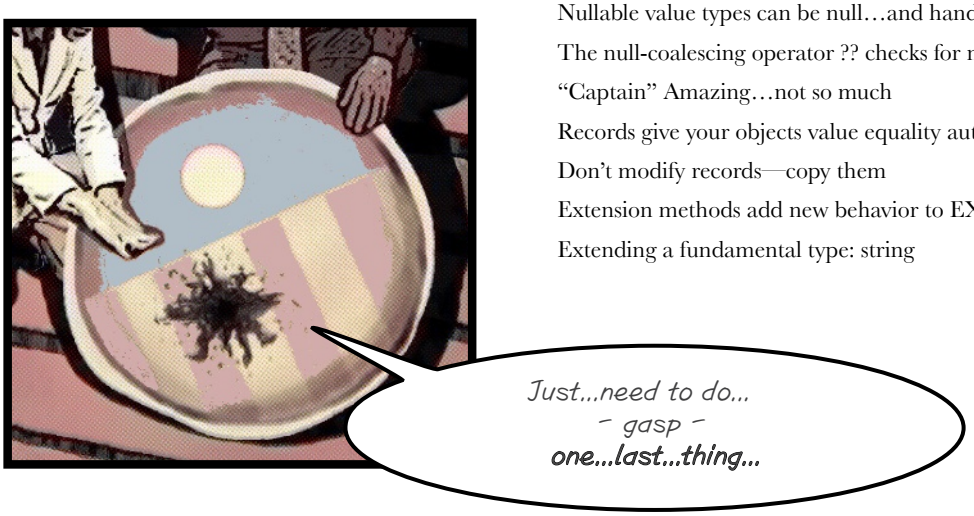
"Captain" Amazing…not so much 715

Records give your objects value equality automatically 717

Don't modify records—copy them 718

Extension methods add new behavior to EXISTING classes 723

Extending a fundamental type: string 724

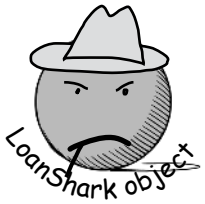


exception handling

12 Putting out fires gets old

When you have to deal with error after error, it's called "putting out fires."

Imagine: it's a few years from now. You spent all that time working on your C# skills and continuing to learn and improve, and now you're one of the most senior developers at a big tech company. Now you're getting panicked phone calls in the middle of the night from your support team because **your app crashes**, or **doesn't behave like it's supposed to**. You want to spend your time writing code, not putting out fires! Because nothing pulls you out of the programming groove like having to fix a strange bug you don't recognize. Luckily, C# has **exception handling**, which lets you write code to **deal with problems** that come up. Better yet, you can even plan for those problems, and **keep things running** when they happen.



Your hex dumper reads a filename from the command line	732
When your program throws an exception, the CLR generates an Exception object	736
All Exception objects inherit from System.Exception	737
There are some files you just can't dump	740
What happens when a method you want to call is risky?	741
Handle exceptions with try and catch	742
Use the debugger to follow the try/catch flow	743
Catch-all exceptions handle System.Exception	745
Use the right exception for the situation	750
Exception filters help you create precise handlers	754
The worst catch block EVER: catch-all plus comments	756
Temporary solutions are OK (temporarily)	757
Use NuGet to add a logging library to your app	759
Add logging to your ExceptionExperiment app	760


```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```

Unity Lab 6

Scene Navigation

In the last Unity Lab, you created a scene with a floor (a plane) and a player (a sphere nested under a cylinder), and you used a NavMesh, a NavMesh Agent, and raycasting to get your player to follow your mouse clicks around the scene. In this lab, you'll add to the scene with the help of C#.

Let's pick up where the last Unity Lab left off	764
Add a platform to your scene	765
Use bake options to make the platform walkable	766
Include the stairs and ramp in the NavMesh	767
Make the player navigate around the obstacles	769
Get creative!	770

Are you ready?

how to use this book

Intro

In this section, we answer the burning question:
"So why DID they put that in a book on learning C#?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you want to **learn C#** (and pick up some knowledge of game development and Unity along the way)?
2. Do you like to tinker? Do you learn by doing, rather than just reading?
3. Do you prefer **interesting and stimulating conversation** to **dry**, **dull**, **academic lectures**?

This book is for you.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Are you more interested in theory than practice?
2. Does the idea of doing projects and writing code make you bored and a little twitchy?
3. Are you **afraid to try something different**? Do you think a book about a serious topic like development needs to be serious all the time?

you might consider trying another book first.

Do I need to know another programming language to use this book?

A lot of people learn C# as a second (or third, or sixteenth) language, but you don't need to have written a lot of code to get started.

If you've written programs (even small ones!) in *any* programming language, taken an introductory programming class at school or online, done some scripting, used a database query language, done some web page design, or worked with Excel functions, then you've ***definitely*** got the background for this book, and you'll feel right at home.

What if you have less experience? That's OK! You can go at your own pace, which is why thousands of beginners have used our book to learn C# as their first programming language.

Still not sure if this book is right for you? Try doing the first few chapters—you can download a free PDF of the first four chapters from https://github.com/head-first-csharp/fifth-edition. If you're still comfortable after doing all of the exercises, then this book will work well for you.

We know what you're thinking

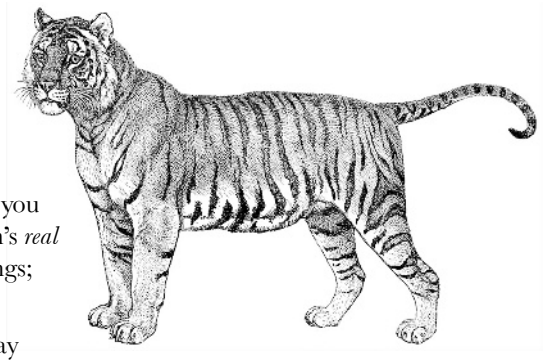
"How can *this* be a serious C# programming book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking

Your brain thinks THIS is important.



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps out in front of you. What happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge.*

And that's how your brain knows…

This must be important! Don't forget it!

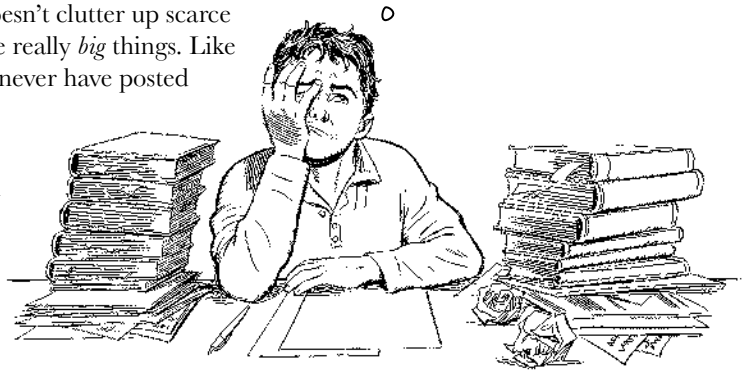
Your brain thinks THIS isn't worth saving.

Great. Only **800** **more** dull, dry, boring pages.

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.

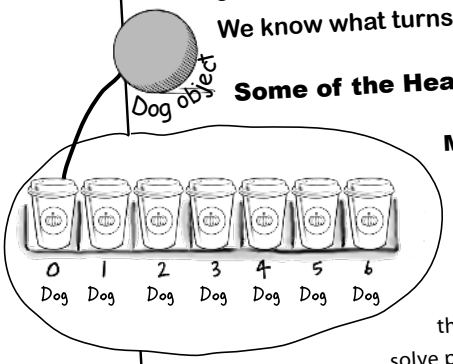
Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page.

And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.



Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). They also make things more understandable.

Put the words within or near the graphics they relate to, rather than at the bottom or on another page, and learners will be up to *twice* as likely to be able to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on postlearning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. Unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?" and the amazing "Aha! I got this!" feeling that comes when you solve a puzzle, learn something everybody else thinks is hard—or maybe just realize you've learned so much *great new stuff* and it feels so good to be able to use it.

Even scary emotions can help ideas stick in your brain.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to build programs in C#. And you probably don't want to spend a lot of time on it. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat C# like it was a hungry tiger?

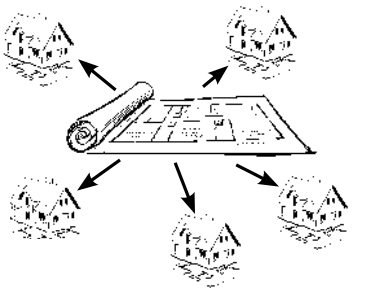
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important, but they keep looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity***, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else on the page, like in a caption or in the body text) causes your brain to try to make sense of how the words and pictures relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning.

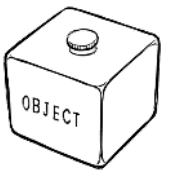
Here's what WE did:



We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***



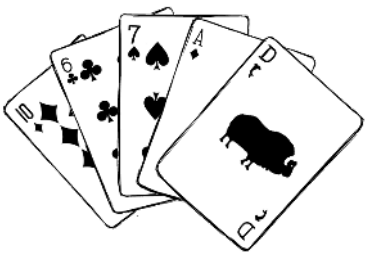
We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included dozens of ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the paper puzzles and code exercises challenging yet doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

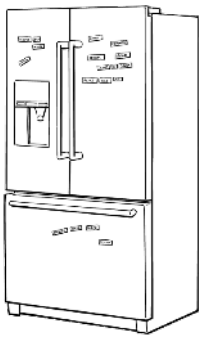
We included content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view***, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.



We included ***challenges***, with exercises, and asked ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are no Dumb Questions" sections.**

   That means all of them. They're not optional sidebars—***they're part of the core content***! Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Write a lot of code!**

   There's only one way to *really* learn C# so it sticks: **write a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. If you get stuck, don't be afraid to **peek at the solution**! We included a solution to each exercise for a reason: it's easy to get snagged on something small. But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

README.md

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

The activities are NOT optional.

The puzzles and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some for understanding, and some to help you apply what you've learned. ***Don't skip the written problems.*** The pool puzzles are the only things you don't *have* to do, but they're good for giving your brain a chance to think about twisty little logic puzzles—and they're definitely a great way to really speed up the learning process.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

Do all the exercises!

The one big assumption that we made when we wrote this book is that you want to learn how to program in C#. So we know you want to get your hands dirty right away, and dig right into the code. We gave you a lot of opportunities to sharpen your skills by putting exercises in every chapter. We've labeled some of them "Do this!"—when you see that, it means that we'll walk you through all of the steps to solve a particular problem. But when you see the Exercise logo with the running shoes, then we've left a big portion of the problem up to you to solve, and we gave you the solution that we came up with. Don't be afraid to peek at the solution—**it's not cheating**! But you'll learn the most if you try to solve the problem first.

We've also included all the exercise solutions' source code with the rest of the code from this book.

You can find all of it on our GitHub page: *https://github.com/head-first-csharp/fifth-edition*

The "Brain Power" questions don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience is for you to decide if and when your answers are right. In some of the Brain Power questions you will find hints to point you in the right direction.

We're targeting C# 12, Visual Studio 2022, and Visual Studio Code.

This book is all about helping you learn C#. The team at Microsoft that develops and maintains C# releases updates to the language. **C# 12** is the current version at the time this book is going into production. We also lean very heavily on Visual Studio, Microsoft's integrated development environment (IDE), as a tool for learning, teaching, and exploring C#. The screenshots in this book were taken with the **latest versions of Visual Studio 2022 and Visual Studio Code** available at the time of production. We included instructions for installing Visual Studio and Visual Studio Code in Chapter 1.

C# is constantly evolving, and new versions will have great new features! But don't worry—the features of C# that are part of the core learning in this book will be unchanged, so you will be able to use this book with future versions of C#. The Microsoft teams that maintain Visual Studio and Visual Studio for Mac routinely release updates, and very rarely those changes will affect screenshots in this book.

The Unity Lab sections in this book target **Unity 2023.2**, the latest version of Unity available with long-term support when this book went into production. We included instructions for installing Unity in the first Unity Lab.

You can watch ***full video walkthroughs*** of Visual Studio, Visual Studio Code, and Unity installation on our YouTube channel: *https://www.youtube.com/@headfirstcsharp*

All of the code in this book is released under an open source license that lets you use it for your own projects. You can download it from our GitHub page (*https://github.com/head-first-csharp/fifth-edition*).

Game Design...and Beyond

How we use games in this book

You're going to be writing code for lots of projects throughout this book, and many of those projects are games. We didn't do this just because we love games. Games can be ***effective tools for learning and teaching C#***. Here's why:

- Games are **familiar**. You're about to immerse yourself in a lot of new concepts and ideas. Giving you something familiar to grab onto can make the learning process go more smoothly.
- Games make it easier to **explain projects**. When you do any of the projects in this book, the first thing you need to do is understand what we're asking you to build—and that can be surprisingly difficult. When we use games for our projects, that makes it easier for you to quickly figure out what we're asking and dive right into the code.
- Games are **fun to write**! Your brain is much more receptive to new information when you're having fun, so including coding projects where you'll build games is, well, a no-brainer (excuse the pun).

We use games throughout this book to *help you learn broader C# and programming concepts*. They're an important part of the book. You should do all of the game-related projects in the book, even if you're not interested in game development. *(The Unity Labs are optional, but strongly recommended.)*

The technical review team

Charlotte M. Ellett

Photo credit: Ryan Lash/TED

Layla Porter

Gerald Versluis

"If I have seen further, it is by standing on the shoulders of giants."
–Isaac Newton

The book you're reading has very few errors in it, and we give a TON of credit for its high quality to our amazing team of technical reviewers—the giants who kindly lent us their shoulders.

To the review team: we're so incredibly grateful for the work that you all did for this book. Thank you so much!

Charlotte M. Ellett is a game developer and programmer, electronics enthusiast and maker, and has been a TED Resident, Red Hat developer relations expert, and Space Camp Crew Trainer. She enjoys making DIY projects and conducting experiments with (not on) her family, and teaching the next generation of engineers and scientists. You can find her at *CharlotteMEllett.com*.

Layla Porter is an experienced software engineer and developer advocate specialising in .NET technologies. Layla has been awarded the Microsoft MVP award for the past five years and the GitHub Star award for the past three. She has spoken at developer conferences all over Europe and North America and is a YouTube content creator. She enjoys hiking, reading urban fantasy novels, playing video games with her husband, and doting on her two miniature pinschers, Cookie and Lily.

Layla actually used the third edition of this book to learn C# in 2015 before applying for her first developer job!

Gerald Versluis is a senior software engineer at Microsoft on the .NET MAUI team. Besides his day job he is also passionate about sharing his knowledge and learning with others. Gerald regularly speaks at conferences, writes blogs and articles, post videos on his YouTube channel, and contributes to open source projects. You can find Gerald on every major social media platform under @jfversluis or on his website: *https://jfversluis.dev*

We also want to give special thanks to **Joe Albahari** and **Jon Skeet** for their incredible technical guidance and really careful and thoughtful review of the first edition, which truly set us up for the success we've had with this book over the years. We benefited so much from your input—even more, in fact, than we realized at the time.

"Shoulders of giants"

This is the **fifth edition** of *Head First C#*, which means a lot of what you'll read in this book was also in an earlier edition. Some of it even dates back to the very first edition from 2008! So it's important to us that we acknowledge the amazing contributions of our **fourth edition tech review team**:

Lindsey Bieda

Ashley Godbold

Tatiana Mac

Lisa Kellner

← Lisa was also on the team for the third, second, and first editions. Thank you!

And our **third edition tech review team**:

Johnny Halife

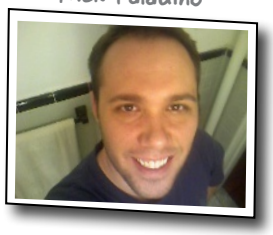
Rebeca Dunn-Krahn

Chris Burrows

And our **second edition team**:

David Sterling

Nick Paladino



Chris and David helped us with both the third and second editions. Thank you!

And finally, our **first edition team**:

Lisa Kellner

Joe Albahari

Jay Hilyard

Daniel Kinnaer

Krishna Pala

Giuseppe Turitto

Aayam Singh

Theodore Casser

Andy Parker

Peter Ritchie

Bill Mietelski

Folks from the first edition team who aren't pictured, but just as amazing: Wayne Bradney, Dave Murdoch, and especially Bridgette Julie Landers.

And super special thanks to our wonderful readers—especially Alan Ouellette, Jeff Counts, Terry Graham, Sergei Kulagin, Willian Piva, Jim Cupec, Mike Olgren, David Horvitz, and Greg Combow—who let us know about issues they found while reading our book, and professor Joe Varrasso at Mohawk College for being an early adopter of our book for his course.

Thank you all so much!!

These people are amazing and it's been such a privilege working with them. Thank you all!

Michele Cronin

Acknowledgments

Our editor:

First and foremost, we want to thank our amazing editor, **Michele Cronin**, for everything you've done for this book. It was such a pleasure working with you!

The O'Reilly team:

Katherine Tozer

There are so many people at O'Reilly we want to thank that we hope we don't forget anyone! First, last, and always, we we want to thank **Mary Treseler**, who's been with us on our journey with O'Reilly from the very beginning. Special thanks to production editor **Katherine Tozer**, indexer **Joanne Sprott**, and **Elizabeth Oliver** for her sharp proofread—all of whom helped get this book from production to press in record time. A huge and heartfelt thanks to **Amanda Quinn**, **Zan McQuade**, **Olivia MacDonald**, **Melissa Duffield**, and **Brian Guerin** for getting this project on track from the beginning, and being so supportive of our books and training courses over the years. And a big shout-out to our other friends at O'Reilly: **Mike Hendrickson**, **Lindsay Ventimiglia**, **David Michelson**, **Jeff Bleiel**, and, of course, **Tim O'Reilly**. If you're reading this book right now, then you can thank the best publicity team in the industry: **Marsee Henon**, **Kathryn Barrett**, and the rest of the wonderful folks at Sebastopol.

We also want to give a shout-out to some of our favorite O'Reilly authors:

- **Dr. Paris Buttfield-Addison**, **Jon Manning**, and **Tim Nugent**, whose book *Unity Game Development Cookbook* is just simply amazing, and whose other O'Reilly books, including *Head First Swift*, are also fantastic.
- **Joseph Albahari,** who wrote the thoroughly indispensable *C# 12 in a Nutshell*.
- **David Pine,** whose *Learning Blazor: Build Single-Page Apps with WebAssembly and C#* is perfect for anyone who wants to build modern web apps in C#.

...and a few more folks who have been so incredibly supportive over the years

Thank you so much to **Cathy Vice** for her amazing piece on epilepsy that we used in Chapter 10, and for all her epilepsy advocacy work. Thank you to **Jesse Liberty** for some great advice and all of his books over the years. And we really, really want to thank **Jeremy Sinclair** for his advice on characters, and for being a pillar of the .NET community.

And an ***enormous thank you to our friends at Microsoft*** who helped us so much with this book—your support through this project was amazing. Thank you so much to **Maddy Montaquila**, **Shane Neuville**, and **Gerald Versluis** from the MAUI team—especially Shane's feedback on our code and Gerald's unbeliavably valuable tech review. We're grateful for **Jen Gentleman** for teaching us about some great Windows features (like the emoji panel!)

We're also so grateful to **Dominic Nahous**, **Jordan Matthiesen**, and **John Miller** from the Visual Studio for Mac team, and to **Cody Beyer**, who was instrumental in getting our whole partnership with that team started. Thank you to **David Sterling** for an awesome review of previous editions, and **Immo Landwerth** for helping us nail down topics we should cover in the fourth edition. ***Extra special thanks*** to **Mads Torgersen**, program manager for the C# language, for all the wonderful guidance and advice he's given us over the years. You all are fantastic!

And finally, we're especially grateful to **Jon Galloway**, who provided so much amazing code for the Blazor projects in the Blazor downloadable PDF—collaborating with you was a truly wonderful experience. Thank you so much!

1 start building apps with C#

Build something great...fast!

Want to build great apps…right now?

With C#, you've got a modern programming language and a **valuable tool** at your fingertips. And with **Visual Studio**, you've got an amazing development environment with highly intuitive features that make coding as easy as possible. Not only is Visual Studio a great tool for writing code, it's also a **really effective learning tool** for exploring C#. Sound appealing? ***Let's get coding!***

Learn C#...and learn to become a great developer

Do you want to become a great developer? Yes? Then you came to the right book! You ***can*** become a great developer, and **C# is the perfect language** to help you get there. Here's why:

- **C# is a *powerful, modern language*** that lets you do incredible things. You can use it to build everything from games to websites to serious business applications. You name it, C# can do it.
- **C# skills are *in demand.*** Are you looking to land a programming job? C# is one of the most in-demand programming languages around because companies all over the world use C# to build their desktop applications and websites.
- **C# is *cross-platform*.** You can write apps that run on Windows, macOS, Linux, and even on your Android and iPhone devices.

...with a learning system that's effective and fun

When you learn C#—when you learn to *really* be effective with it—you're learning more than just a language. You're learning a ***whole new way of thinking***… and that's where we come in. We've spent more than 15 years developing, experimenting with, and testing out new and different ways to help you get C# ideas into your brain. You'll use **powerful programming environments** to build real projects and write lots of code. You'll learn and practice important **development ideas and patterns** that help you write great code. You'll learn how to use **modern AI tools** to superpower your code and level up your learning skills. By the time you're done, you'll have the foundation for successful and satisfying software development.

Welcome to the world of C#. Let's dive in!

Many people who used previous editions of this book have reached out to us over the years to tell us how our book helped them start their development careers. We're looking forward to hearing from you too!

"Head First C# started my career as a software engineer and backend developer. I am now leading a team in a tech company and an open source contributor."

—Zakaria Soleymani, Development Team Lead

"Thank you so much! Your books have helped me to launch my career."

—Ryan White, Game Developer

Write code and explore C# with Visual Studio

The best way to get started with C# is to **write lots of code**.

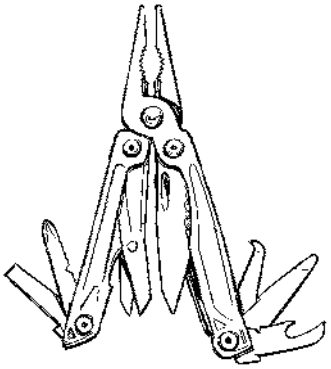
This book uses **pictures, puzzles, quizzes, stories, and games** to help you learn C# in a way that suits your brain. Every one of those elements is built to help you with a single goal: to keep things interesting while we help you get C# concepts, ideas, and skills into your brain.

This book is also **full of C# projects** that are ***specifically designed*** to give you lots of different ways to explore C# and learn about important ideas and concepts that will help you become a great developer. We designed those projects to be engaging, fun, and interactive to give you lots of opportunities to put those concepts, ideas, and skills into practice.

Visual Studio is only available for Windows. Luckily, you can also use *Visual Studio Code* to do all of the projects in this book. If you're using a Mac or Linux, or if you want to use Visual Studio Code instead of Visual Studio, skip ahead to the next section, where we'll show you how to set it up and use it to create your first C# project.

Visual Studio is your free gateway to C#

Learning C# is all about exploring and growing your skills ***at your own pace***, and that's where **Visual Studio** comes in. It's an amazing tool built by Microsoft. At its heart, it's an editor for your C# code and projects, but it's much more than that. It's a creative tool that helps you with every aspect of C# development. We'll use Visual Studio throughout this book as an important tool to help you learn and explore C#.



Visual Studio is an **IDE**—that's short for ***integrated development environment***—a text editor, visual designer, file manager, and debugger…it's like a multitool for everything you need to write code.

Here are just a few of the things that Visual Studio helps you do:

- **It's a file and project manager.** C# projects are often made up of a lot of files. Visual Studio makes it easy to see exactly where they are, and integrates with version control systems like Git to make sure you never lose a line of code.
- **It helps you edit and manage your code.** Visual Studio has many intuitive features to help you edit your code and C# projects, including powerful AI-driven tools like IntelliSense pop-ups and IntelliCode code completion that give you great suggestions to help keep you in the flow.
- **It's a debugger that lets you see your code in action.** When you debug your apps in Visual Studio, you can see exactly what your code is doing while it runs—which is a great way to ***really understand*** how C# code works.

Visual Studio is a powerful development environment, and it's an amazing learning tool to help you explore C#.

IDE Tips

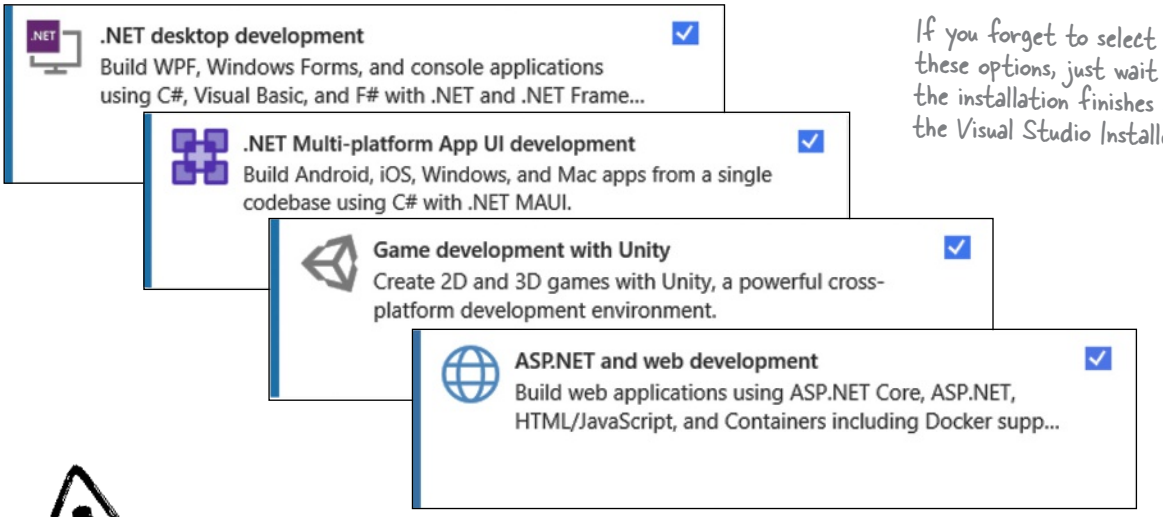
We'll often refer to Visual Studio as **"the IDE"** throughout this book. Keep an eye out for handy IDE tips that help you become a more efficient coder.

If you decide to use Visual Studio Code instead of Visual Studio, that's your IDE. They're both IDEs!

Install Visual Studio Community Edition

Open *https://visualstudio.microsoft.com* and **download Visual Studio Community Edition**. It's available for both Windows and macOS. The installers look a little different depending on which platform you're using. Make sure you install the .NET desktop development tools and .NET Multi-platform App UI (or .NET MAUI) development tools. We'll be doing 3D game development with Unity, so make sure you check that option too.

When you run the Visual Studio installer, select the ".NET desktop development," ".NET Multi-platform App UI development (MAUI)," and "Game development with Unity" options to install the Visual Studio tools you'll use in this book. You should also select "ASP.NET and web development" if you plan to download the **Head First C# Blazor Learner's Guide** and learn about web development with C#.



If you forget to select one of these options, just wait until the installation finishes and run the Visual Studio Installer again.

This book was written on a Mac. All of the Visual Studio and Unity screenshots in this book were taken running Windows 11 in a Parallels Desktop virtual machine.

Watch it!

You can use Visual Studio for Mac, but it's no longer supported by Microsoft.

Visual Studio for Mac was a great tool for learning C# on a Mac! But while we were finishing up this edition of Head First C#, Microsoft announced that they would stop development on it and end support in 2024. If you're using a Mac or Linux, the folks at Microsoft recommend running Windows virtual machine (VM) using software like VirtualBox or Parallels Desktop.

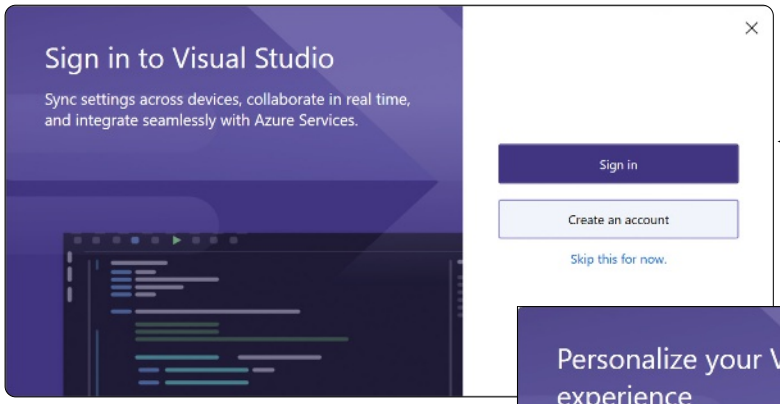
An easy way to get started is to download VirtualBox for free from https://www.virtualbox.org and then download a Windows virtual machine from Microsoft that contains an evaluation version of Windows from https://developer.microsoft.com/en-us/windows/downloads/virtual-machines.

*We'll also show you how to use **Visual Studio Code** later in the chapter.*

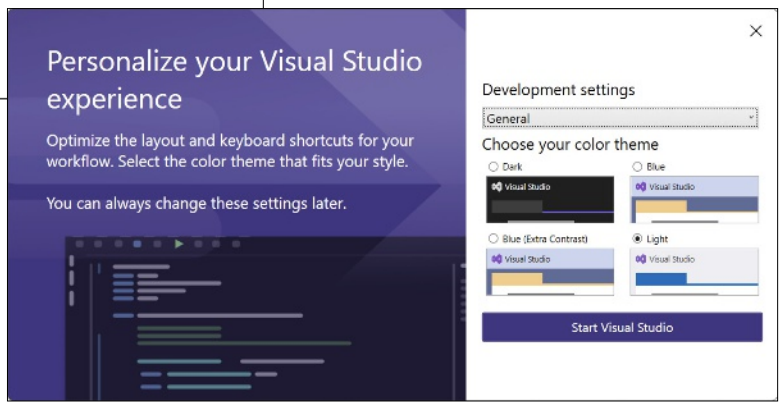
All of the projects in this book can be done in Visual Studio Code on Windows or macOS.

Run Visual Studio

We're going to jump right into code! Once the installer finishes, **run Visual Studio**.



The first time you start Visual Studio, it may ask you to sign in to your Microsoft account. If you don't have an account, you can create one by clicking the "Create an account" button. This is how Microsoft activates your free license for the Community Edition of Visual Studio.



You'll also be prompted to choose a color theme. We used Light for the screenshots in this book because they're easier to see in print, but many users prefer the Dark theme.

Keep an eye out for these "relax" boxes—they point out some common issues that a lot of readers run into, so you know they're coming and don't have to worry about them.

Relax

Grab a cup of coffee—it can take some time for Visual Studio to install.

Don't worry if it takes a few minutes (or more!) to finish installing Visual Studio. And while we're on the subject, here's something else that you don't have to worry about.

All of the screenshots in this book were taken with **Visual Studio 2022 Community Edition**, the latest version available while we were writing it. If Microsoft released a newer version of Visual Studio since we took these screenshots, feel free to try it! The code and ideas that we teach should still work just fine. But if you want the screenshots to match, Microsoft makes older versions of Visual Studio available for download—and you can always install different versions of Visual Studio on the same computer: *https://visualstudio.microsoft.com/vs/older-downloads*

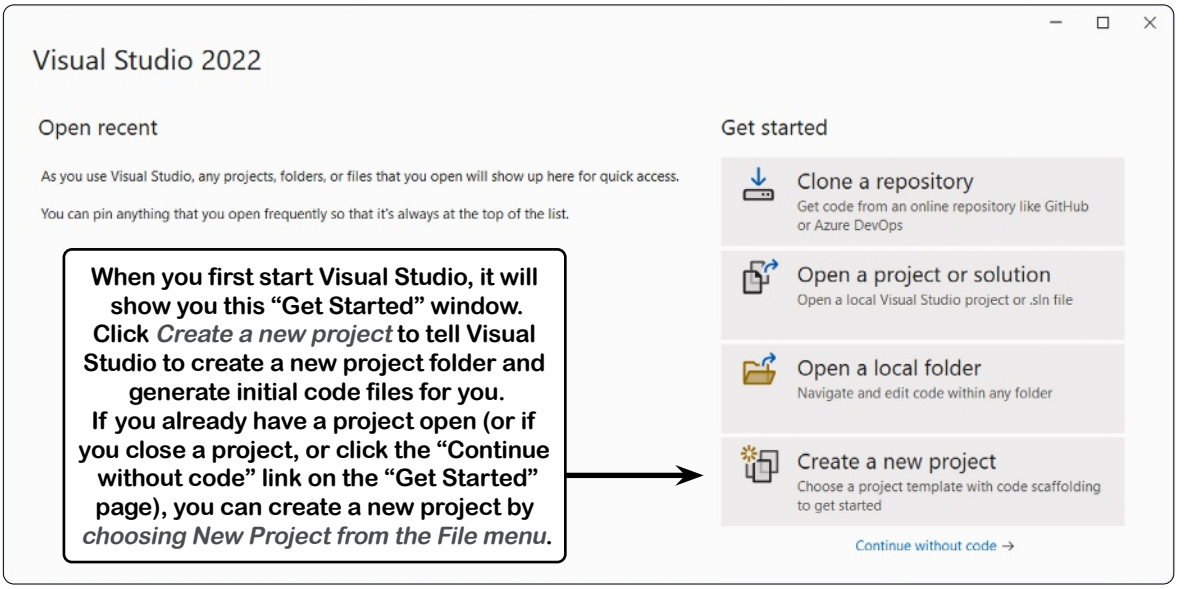
If you run into trouble installing Visual Studio or running your first project, head to our YouTube channel (*https://www.youtube.com/@headfirstcsharp*) to see videos of the entire installation process.

Create and run your first C# project in Visual Studio

The best way to learn C# is to start writing code, so you're going to write a lot of code—and create a lot of apps!—throughout this book. Each app will get its own **project**, or a folder that Visual Studio creates with special files to organize all of the code.

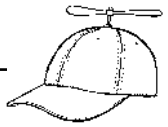
1 Tell Visual Studio to create a new project.

When you launch Visual Studio, the first thing you'll see is a Get Started window with four options. **Click "Create a new project"** to create a new project.



When you first start Visual Studio, it will show you this "Get Started" window. Click *Create a new project* to tell Visual Studio to create a new project folder and generate initial code files for you. If you already have a project open (or if you close a project, or click the "Continue without code" link on the "Get Started" page), you can create a new project by *choosing New Project from the File menu*.

When you create a new project, Visual Studio will ask you which of its **project templates** you want to use. Every C# project consists of a set of folders and files. Visual Studio has many built-in templates that it can use to generate different kinds of projects. In this book, you'll use Visual Studio's templates to create three kinds of projects: Console App projects, .NET MAUI projects, and MSTest unit test projects. (You'll also create Unity projects, but you won't use Visual Studio to create them.)



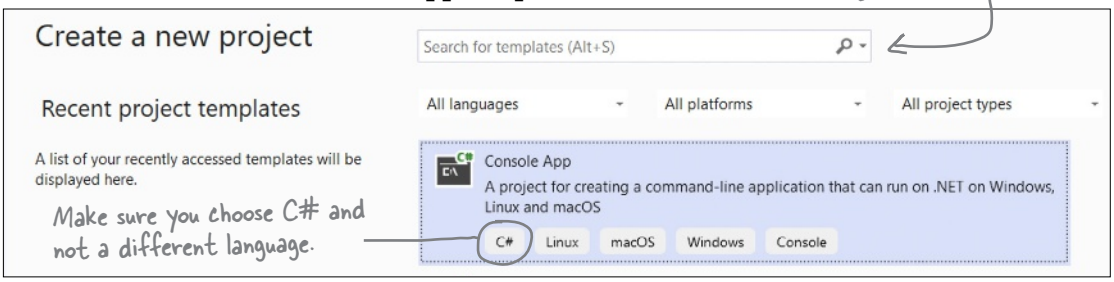
Let's write some code!

You'll be writing a lot of code throughout this book, which means you'll be **creating a lot of projects**. Most of those projects will be **Console App projects**, just like the one you're creating now—so you can follow these directions any time you need to create a new Console App project. Just make sure you choose a different project name each time so that Visual Studio creates the project in a new folder (don't worry—it will warn you if that name already exists).

2 Choose a project template for Visual Studio to use.

Visual Studio creates new projects using a *template* that determines what files to create. **Choose the Console App template** and click Next.

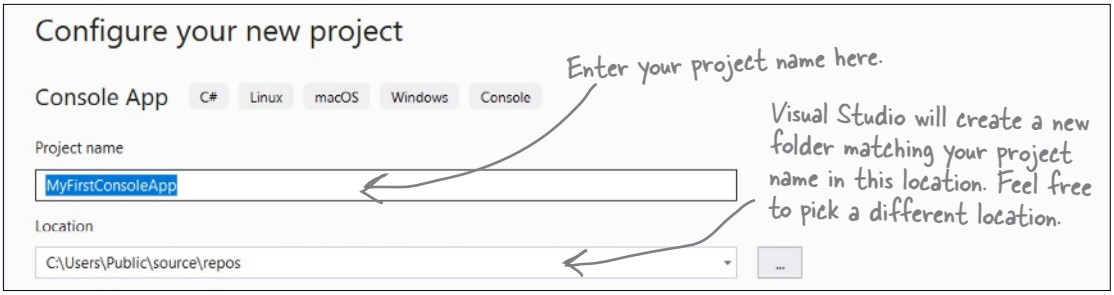
Enter "Console App" in the search box or scroll down to the Console App template.



Make sure you choose C# and not a different language.

3 Enter a name for your project and click Next.

Your project's name is important—it determines file and folder names, and you'll see it inside some of the code that Visual Studio generates for you. If we ask you to pick a specific name, make sure you do; otherwise, the code in your project may not match screenshots in the book.

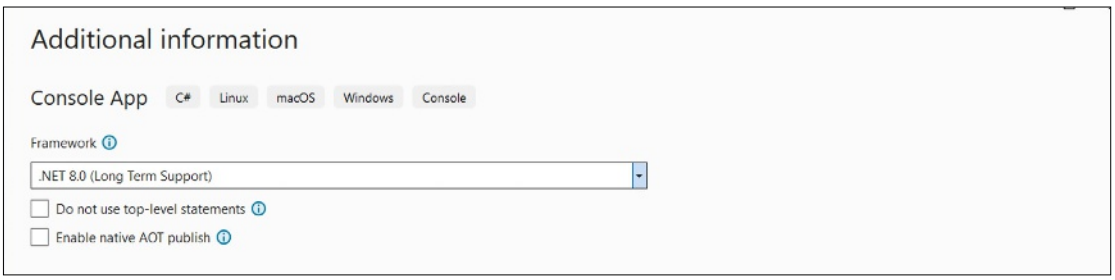


Enter your project name here.

Visual Studio will create a new folder matching your project name in this location. Feel free to pick a different location.

4 Make sure you're using the current version of .NET.

The current version of .NET at the time we're writing this is 8.0—make sure the version that you're using is 8.0 (or higher). Then **click the Create button** to create your project.



Once Visual Studio creates your project, it will open a file called *Program.cs* with this code:

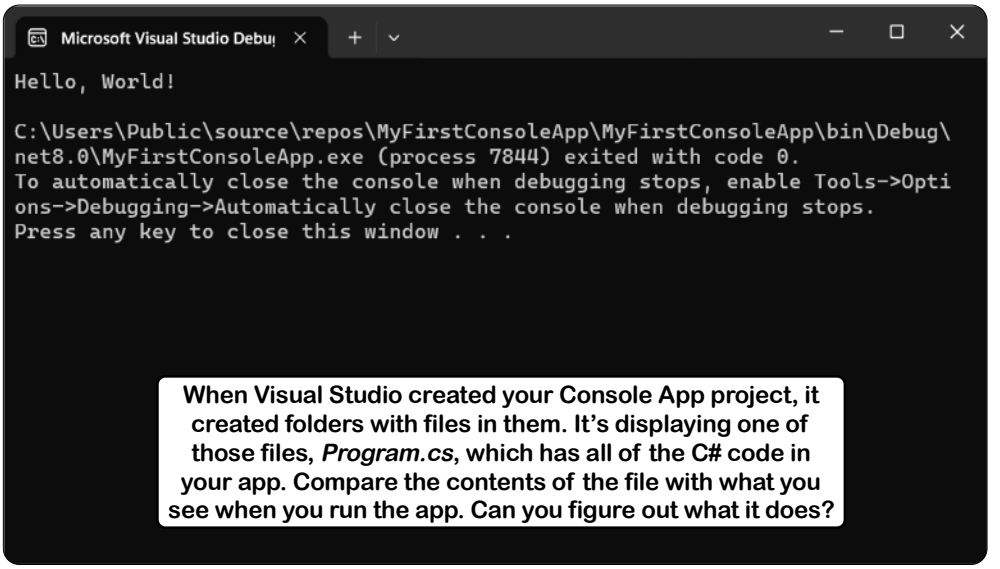
The code for your app is in a file called *Program.cs*. You can edit it in this window.

5 **Run your app.**

The app Visual Studio created for you is ready to run. At the top of the Visual Studio IDE, find the button with a green triangle and your app's name and click it:

6 **Look at your app's output.**

When you run your program, the **Microsoft Visual Studio Debug Console window** will pop up and show you the output of the program:



At the top of the window is the **output of the program**:

```
Hello, World!
```

Then there's a line break, followed by some additional text:

```
C:\path-to-your-project-folder\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\
net8.0\MyFirstConsoleApp.exe (process ####) exited with code 0.
To automatically close the console when debugging stops, enable Tools→Options-
>Debugging→Automatically close the console when debugging stops.
Press any key to close this window . . .
```

You'll see the same message at the bottom of every Debug Console window. Your program printed a single line of text (`Hello, World!`) and then exited. Visual Studio is keeping the output window open until you press a key to close it so you can see the output before the window disappears.

Press a key to close the window. Then run your program again.

This is how you'll run all of the Console App projects that you'll build throughout the book.

Sharpen your pencil

Visual Studio is a great tool to help you learn and explore C#. You're going to be writing a lot of code throughout this book, so this is a great time to get familiar with it.

Visual Studio is an IDE, or integrated development environment. If you haven't used an IDE before it may look very busy, so this is a great time to get used to it. One of the most important ways to get new ideas, information, skills, and tools to stick in your brain is to **write them down**. So take a close look at the different parts of Visual Studio and write down what you think they do. It's OK if you aren't 100% sure—just take your best guess!

.....

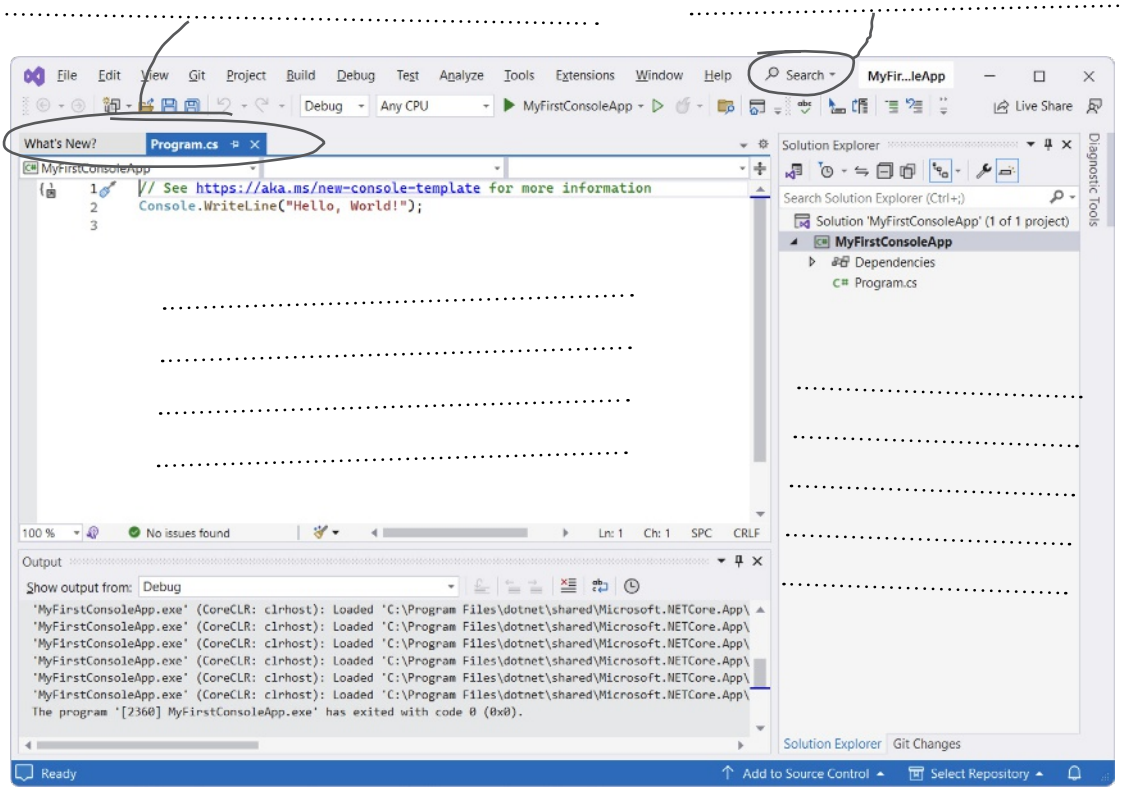
.....

.....

.....

.....

.....



When you created your app, it contained a file with two lines. Write down what you think each line does.

```
// See https://aka.ms/new-console-template for more information.
```

.....

```
Console.WriteLine("Hello, World!");
```

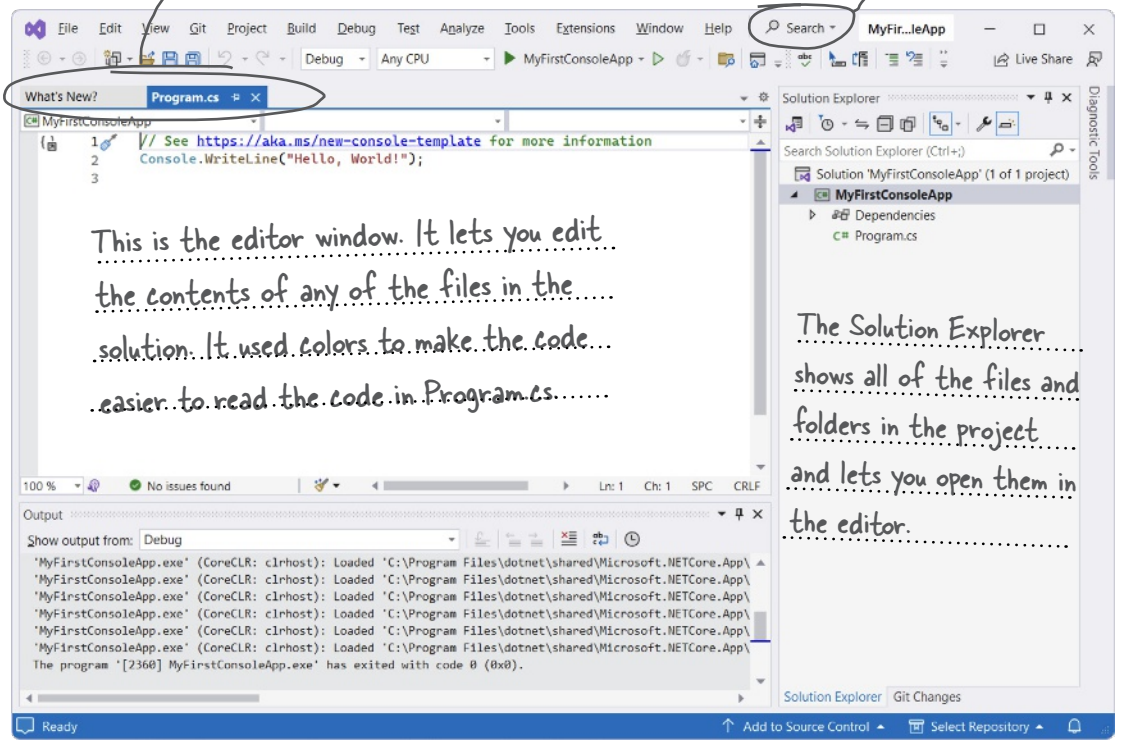
.....

Try moving the panels in Visual Studio around. Click the pushpin button (📌) to collapse the Solution Explorer window into the side panel. Reset the layout by choosing Reset Window Layout from the View menu, then choosing Output from the View menu.

Sharpen your pencil Solution

You can open multiple files at the same time and use these tabs to switch between them. Visual Studio displayed the "What's New?" tab when you opened it.

The Search button opens a window that lets you search the code in your project and find features in Visual Studio.



This is the editor window. It lets you edit the contents of any of the files in the solution. It used colors to make the code easier to read the code in Program.cs.

The Solution Explorer shows all of the files and folders in the project and lets you open them in the editor.

When you created your app, it contained a file with two lines. Write down what you think each line does.

```
// See https://aka.ms/new-console-template for more information.
```

This is a comment, it doesn't do anything. It just provides information to the person reading it.

```
Console.WriteLine("Hello, World!");
```

This line of code causes the app to print the text "Hello, World!" (without the quotation marks).

Keep an eye out for these Q&A sections. They often answer your most pressing questions, and point out questions other readers are thinking of. In fact, a lot of them are real questions from readers of previous editions of this book!

there are no Dumb Questions

Q: So if Visual Studio writes code for me, is learning C# just a matter of learning how to use it?

A: No. The IDE is great at automatically generating some code for you, but it can only do so much. There are some things it's really good at, like setting up good starting points for you, and automatically changing properties of controls in your UI. It did all of that by taking the information you gave it and feeding it into a template that generated files. The most important part of programming—figuring out what your program needs to do and making it do it—is something that no IDE can do for you. Even though the Visual Studio IDE is one of the most advanced development environments out there, it can only go so far. It's ***you***—not the IDE—who write the code that actually does the work.

Q: What if the IDE creates code I don't want in my project?

A: You can change or delete it. The IDE is set up to create code based on the way the element you dragged or added is most commonly used, but sometimes that's not exactly what you wanted. Everything the IDE does for you—every line of code it creates, every file it adds—can be changed, either manually by editing the files directly or through an easy-to-use interface in the IDE.

Q: Why did you ask me to install Visual Studio Community Edition? Are you sure that I don't need to use one of the versions of Visual Studio that isn't free to do everything in this book?

A: There's nothing in this book that you can't do with the free version of Visual Studio (which you can download from Microsoft's website). The main differences between Community and the other editions aren't going to stop you from writing C# and creating fully functional, complete applications. (The paid editions have additional features that are useful for professional software organizations and teams.)

Q: My screen doesn't look like yours! It's missing some of the windows, and others are in the wrong place. Did I do something wrong? How can I reset it?

A: If you click on the **Reset Window Layout** command under the Window menu, the IDE will restore the default window layout for you. Then use the **View** menu to open any windows that are missing. You can find some of the windows you'll see later in this chapter in its **Other Windows** submenu. That will make your screen look like the ones in this chapter and throughout the book.

Some windows collapse by default. Use the pushpin button in the upper-right corner of the window to make it stay open.

Visual Studio will generate code you can use as a starting point for your applications. Making sure the app does what it's supposed to do is entirely up to you.

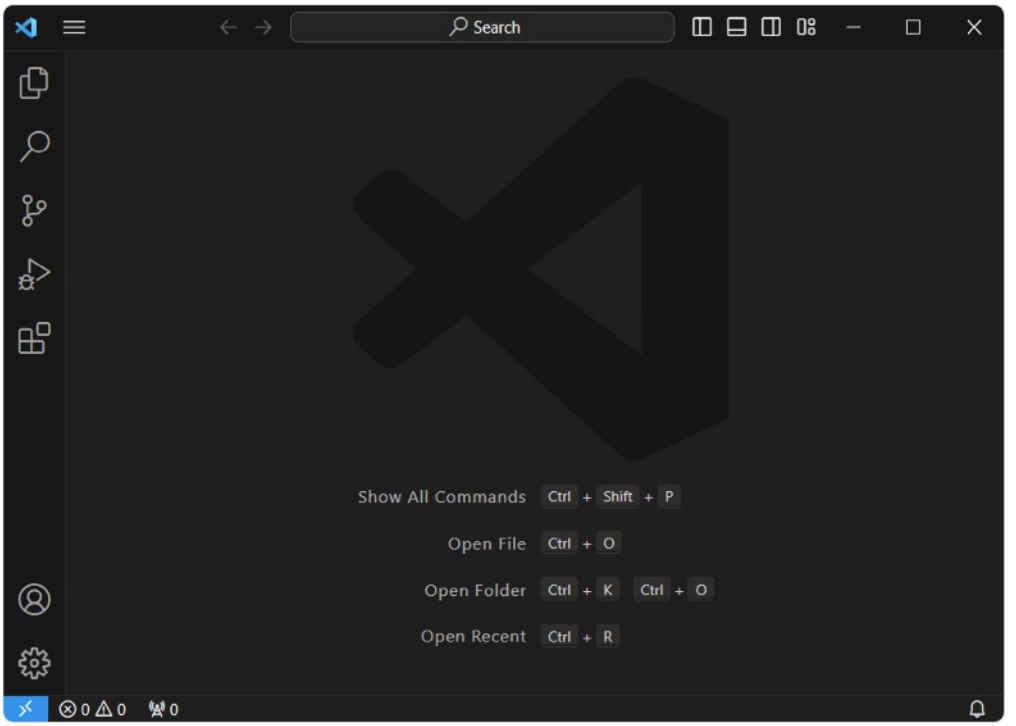
You can use Visual Studio Code with Head First C#

If you're using Visual Studio (and not VSCode), you can *skip ahead* to "Let's Build a Game!"

If you've been around the development world at all over the last few years, you've probably heard a lot of buzz and excitement about **Visual Studio Code** (often called **VSCode**). It's a powerful code editor that runs on Windows, Mac, and Linux, and it's gotten very popular among developers because it's easy to use, versatile, fast, and intuitive.

If you're using Windows, we recommend that you consider using Visual Studio (*not* VSCode) because it's specifically built for C# and has some built-in tools that VSCode currently lacks. However, ***all of the projects in this book can be done with VSCode***. Most of the screenshots in this book will show Visual Studio, but we'll also tell you how to do the same thing in VSCode where it differs from Visual Studio.

To use this book with Visual Studio Code, start by downloading it from *https://code.visualstudio.com*. Run the installer, and choose all of the default options. After the installer finishes, open VSCode. It will prompt you to choose a color theme. We chose Dark Modern for our screenshots because we used a light theme for Visual Studio, so choosing a dark theme for VSCode will help make it easier to tell the screenshots apart.



The first time you run VSCode, it shows you a Welcome tab with a bunch of settings you can choose, including color theme. You can go through those settings, or you can just close the tab.

Using Visual Studio Code is optional. You can use VSCode to do all of the projects in this book on Windows or macOS. (Linux readers may need to run an Android emulator to do the .NET MAUI—we'll talk about that later.)

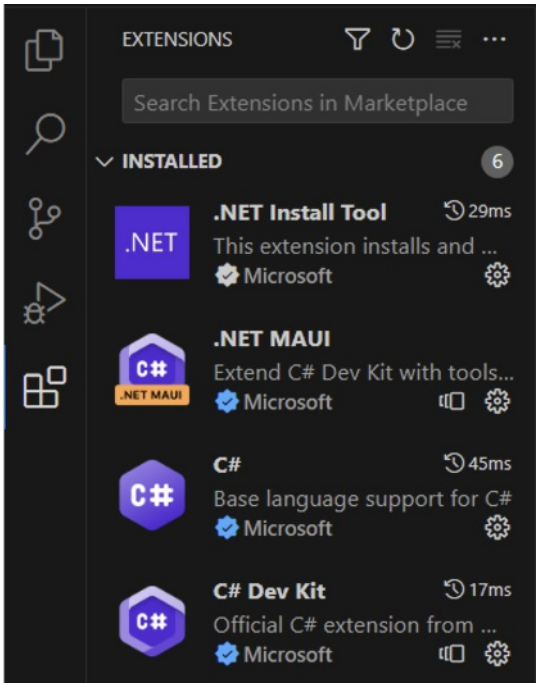
If you have trouble installing VS Code or running your first project, head to our YouTube channel (*https://www.youtube.com/@headfirstcsharp*) to see videos of the entire Windows or macOS installation process.

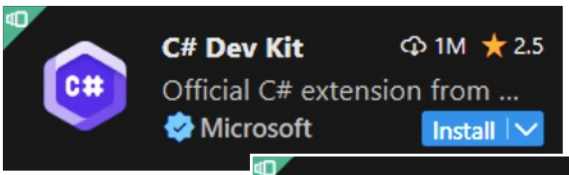
Install the C# extensions

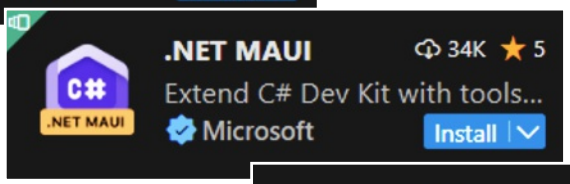
Click the Extensions button on the left side of the VSCode window to open the Extensions panel. At the top of the panel is a search box with the text "Search Extensions in the Marketplace." **Search for each of these extensions**:

- **C# Dev Kit**: This extension has the tools you need to create, edit, and debug C# and .NET projects.
- **.NET MAUI:** Most chapters in this book have a project that uses .NET MAUI, a framework for creating desktop and mobile apps in C#.
- **Unity**: The Unity Labs give you a chance to practice your C# skills by building 3D games and simulations.

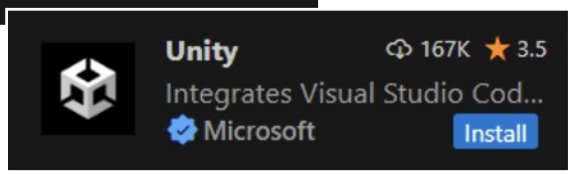
Make sure each extension is the official one from Microsoft. **Click the Install button** on each extension to install it.







VSCode may prompt you to restart after installing extensions. It may also display Getting Started tabs that have useful information.

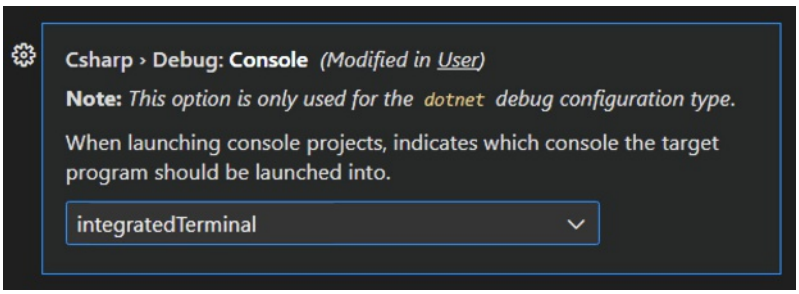


This setting tells VSCode where to show the output of your app. Set it to use the Terminal window; otherwise, some apps you build later in the book won't work.

Change the C# debug console setting

Once you have your extensions set up, click the gear icon in the lower-left corner of the VSCode window and choose Settings (or press Ctrl comma or ⌘ comma). Search for the setting **csharp.debug.console**—you should see a dropdown with several options. Change the setting to **integratedTerminal**.

Now you're ready to write some C# code!

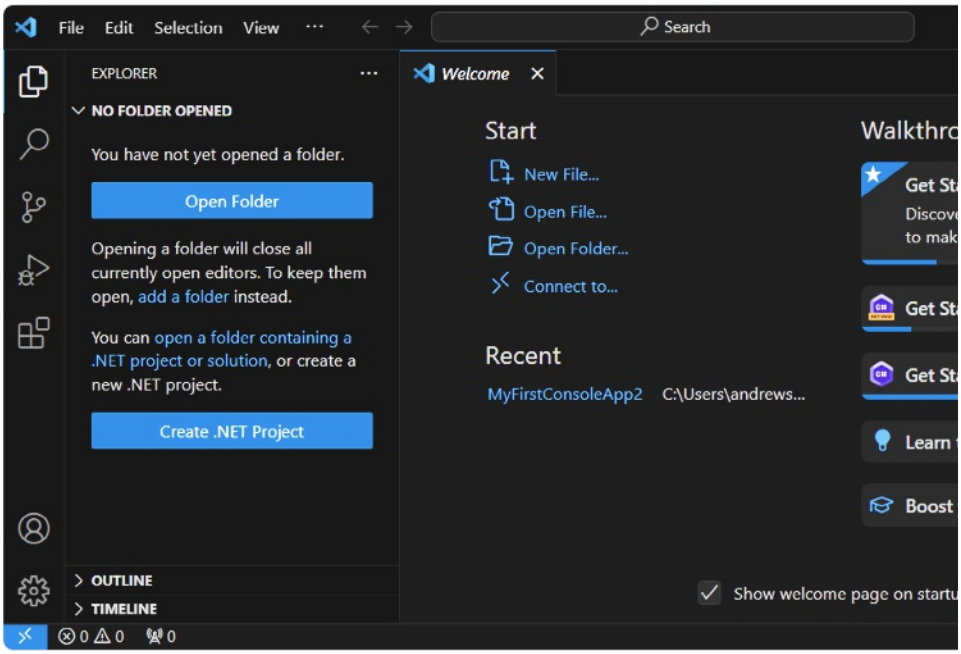


Create and run your first project in Visual Studio Code

Visual Studio Code is first and foremost an **editor**, which means its features are specialized for opening and editing many different kinds of files. A VSCode window is typically used to edit files in a **folder** and its subfolders. When you open VSCode, it remembers the most recent folder you opened. But when you first open it, you'll need to select a folder. We'll walk you through the steps for creating a folder with a new .NET project in it.

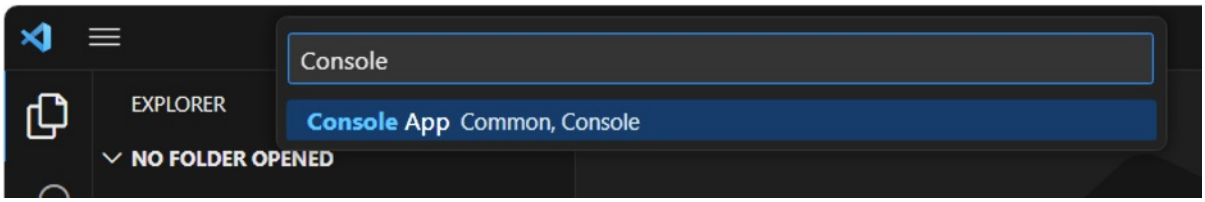
On Windows, the menu is at the top of the window (if you make the window very narrow, click the menu button ☰ to display it). On macOS, you'll see the menu in the menu bar at the top of the screen.

Click this button to open and close the lefthand panel. If VSCode does not currently have a folder open, the panel will have buttons to open a folder and create a new .NET project.



If you don't see the Create .NET Project button, you can open the Command Palette (Ctrl+Shift+P or ⇧⌘ P) and choose .NET: New Project from the menu that's displayed.

1. **Click the Create .NET Project button.**
A box will appear at the top of the VSCode window with a list of project types, and a search box with the prompt "Select a template to create a new .NET Project." **Type `Console` into the search box**, then choose Console App from the list of templates to create a new .NET Console App project.



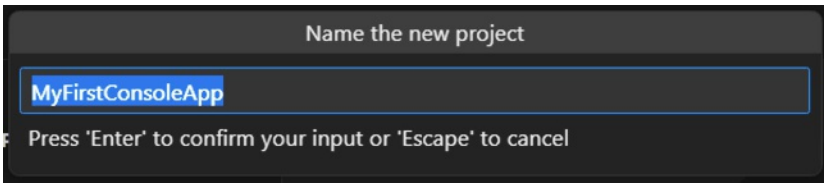
2 Select a folder for your new project.

VSCode will display a folder browser window. **Choose a location for your new project.** The folder browser window has a "New folder" button. You'll be creating a lot of projects throughout this book, so we suggest creating a folder called "Projects" underneath your home folder or Documents folder to hold them.

Create a new folder inside the folder where you keep your projects and **name it *MyFirstConsoleApp*.** Then navigate to the MyFirstConsoleApp folder that you just created and **click Select Folder.**

3 Give your project a name.

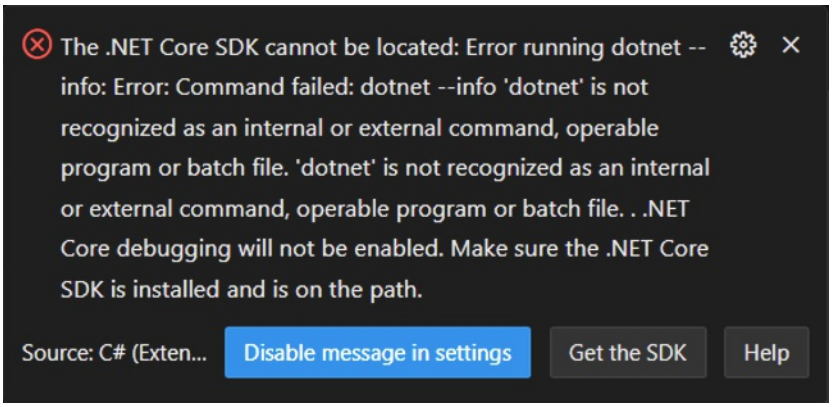
Every C# project has a name. You'll usually give the project folder the same name as the project. After you select your folder, VSCode will prompt you for a project name:



Type `MyFirstConsoleApp` into the box, then press the Enter key to create the project. VSCode may ask you if you trust the authors of the folder. This is a really useful security feature, because it prevents you from accidentally opening malicious code. **Click the "Yes, I trust the authors" button.** You have the option of checking a box to always trust everything in your projects folder.

4 Install the .NET Core SDK. (You only need to do this once!)

Before you can create and run C# and .NET apps, you need to install the **.NET Core SDK**. The easiest way to do this on Windows is to install Visual Studio 2022. If you don't have the SDK installed, VSCode will display a window prompting you to get it. **Click the "Get the SDK" button**—this opens a browser window with the page *https://dot.net/core-sdk-vscode*. Follow the instructions to download the latest version of your SDK for your operating system. Be careful to choose the architecture that matches your computer. For Mac users: if your Mac was made after 2019, choose Arm64; if you're using an older Intel Mac, choose Intel.



Visual Studio Code is really popular with a lot of developers because it's lightweight, open source, and has a huge ecosystem of extensions and tools. But using it will require you to do a little more manual work like this, which is why we recommend using Visual Studio instead.

5 Expand the Solution Explorer and open *Program.cs*.

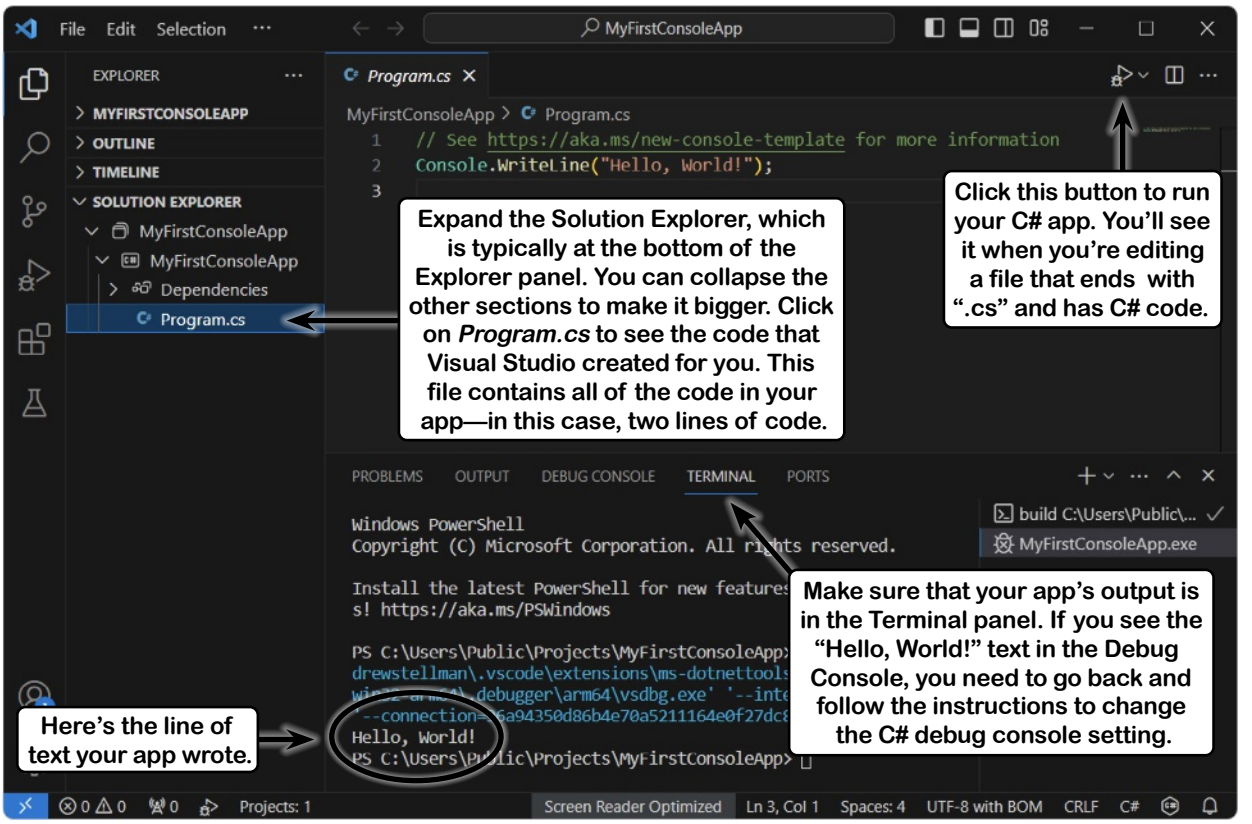
After VSCode creates your C# project, the Explorer panel on the left will contain several collapsible sections. VSCode is file- and folder-based, and the **Explorer** is used to browse those files and open them for editing.

Expand the Solution Explorer section at the bottom of the Explorer. The Solution Explorer is part of the C# Dev Kit, which lets VSCode work with C# projects. It shows you all of the files and subfolders that VSCode created for your project—in this case, your app has one file with C# code called *Program.cs*. **Click *Program.cs*** in the Solution Explorer to open the file.

6 Run your app.

When you have a C# code file (a file that ends with *.cs*) open in the Solution Explorer, you'll see a Run button ( ) in the upper-right corner of the window. Click that button to run the app.

You can also **press F5** and choose "Start Debugging" from the Run menu to run your app. VSCode may prompt you to select a debugger. If it does, choose C#. If it asks for a launch configuration, choose the one that matches the project name. You can press F5 any time you want to run your app.Your app will start running, and Visual Studio will open the **Debug Console panel** to show you the output and let you interact with it. ***This is how you'll run all of the Console App projects that you'll build throughout the book.***



Set up Visual Studio Code for the next project

VSCode is a great code editor! But unlike Visual Studio, it's not designed specifically for C# and .NET projects. It does a great job with C#, but you need to do a little extra work to get it up and running.

First open a folder, then add a project

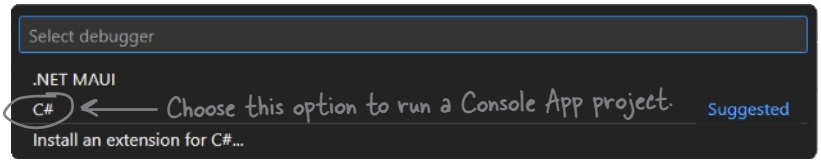
VSCode is extremely flexible, and there are many ways to use it. If you're new to VSCode, we recommend that you **create a new folder for every project in the book**. When you start a new project, **choose Close Folder** from the File menu to close the current folder, then create a new folder and open it.

This is how you reset VSCode so you can start a new project.

The Command Palette

All of the actions that you need to do to create and run projects can be run from the **Command Palette**, the centralized hub for all of the VSCode features. Press Ctrl+Shift+P (or ⇧⌘P on a Mac) to display the Command Palette. Use the .NET: New Project to create a new project in the current folder. There are also commands to open and close .NET solutions. You'll learn more about solutions throughout this chapter.

When you have a .NET app's project folder open, you can run it by choosing **Debug: Start Debugging** from the Command Palette. Choose the C# option to start running your Console App project.



You'll need to install .NET MAUI before you move on to the rest of the chapter. If you're using Visual Studio 2022 you already installed it, but if you're using VSCode you'll need to do it manually.

Install .NET MAUI before reading the rest of this chapter

In the rest of this chapter, you'll build a game using **.NET MAUI** (Multi-platform App UI), a powerful cross-platform framework that lets you create visual apps in .NET and C# that can run on Windows, macOS, Android, and iOS.

Before you can install and run your .NET MAUI apps, you'll need to install the **.NET MAUI workload for .NET**. The easiest way to do this on Windows is to install Visual Studio 2022 and choose the .NET MAUI option.

You can also install .NET MAUI from the command line. Typically it looks like this:

```
dotnet workload install maui        or        sudo dotnet workload install maui
```

If you're using a Mac or Linux, you may need to use `sudo` to run with elevated privileges. If you're on a Mac, you'll also need to **install XCode**. You can install the Android SDK as well (but it's optional). See this page for more details: *https://learn.microsoft.com/dotnet/maui/get-started/installation?tabs=visual-studio-code*

If you're using Linux, you'll need an Android device for the .NET MAUI projects

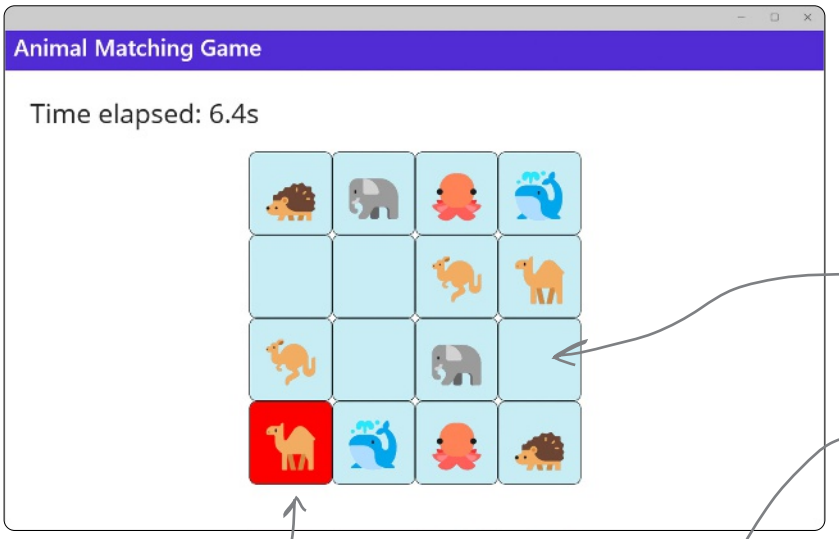
.NET MAUI does not run natively on Linux. If you have an Android device, you can debug directly on it. This page shows you how to set up an Android device so you can connect it to your computer and run your MAUI apps on it: *https://learn.microsoft.com/dotnet/maui/android/device/setup*

As an alternative, every MAUI project in this book has a Blazor alternative, where you'll build a web app version that runs in a browser. Download the **Head First C# Blazor Learner's Guide** from our GitHub page for more information—you can get it as a free PDF: *https://github.com/head-first-csharp/fifth-edition*

Let's build a game!

You've built your first C# app, and that's great! Now that you've done that, let's build something a little more complex. We're going to build an **animal matching game**, where a player is shown eight pairs of animals and needs to click on them in pairs to make them disappear.

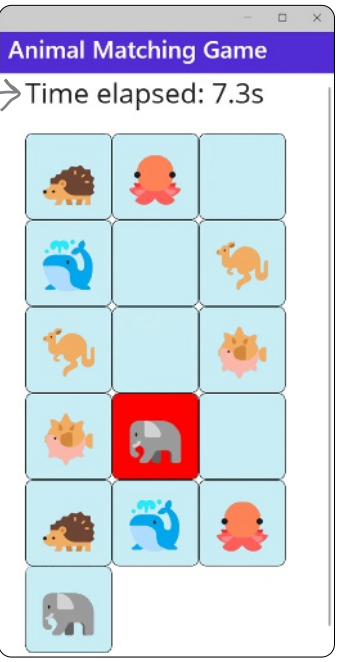
The game displays 16 buttons with eight pairs of matching animals in a random order. You play by clicking animals in pairs: first click one animal, then click its match. Match all eight animals and you win the game!



When you click a matched pair of animals, they both disappear.

When you click the first button, it changes color. If you click on its match, then both animals disappear. If you click any other animal, the color of the first button changes back and you have to start over with a new pair.

To make things more exciting, the game starts a timer as soon as you start the game. Can you beat your best time?



You can change the size of the window and the animal buttons will rearrange themselves out to fill up the new width.

The rest of this chapter is dedicated to walking you through creating the Animal Matching Game project. This project is built with .NET MAUI, a cross-platform framework for building desktop and mobile apps. There are several MAUI projects scattered throughout this book. You also have the option of using these projects to *learn about web development* using Blazor, Microsoft's powerful web development framework. Download the Head First C# Blazor Learner's Guide—it's a free PDF with web versions of every MAUI project in this book (*https://github.com/head-first-csharp/fifth-edition*).

When you've found all eight pairs of animals, the game displays a big "Play again?" button, with your final time underneath it. Click the button to reset the game and start over again!

Keep an eye out for these "Game Design...and Beyond" elements scattered throughout the book. We'll use game design principles as a way to learn and explore important programming concepts and ideas that apply to any kind of project, not just video games.

Game Design...and Beyond

What is a game?

It may seem obvious what a game is. But think about it for a minute--it's not as simple as it seems.

* Do all games have a **winner**? Do they always end? Not necessarily. What about a flight simulator? A game where you design an amusement park? Or a farming simulator? What about a game like The Sims?

* Are games always **fun**? Not for everyone. Some players like a "grind" where they do the same thing over and over again; others find that miserable.

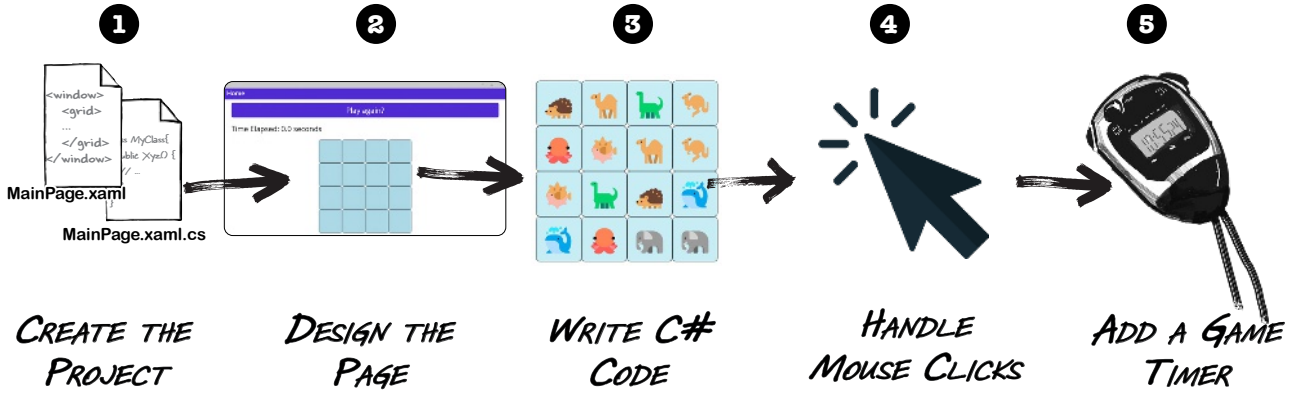
* Is there always **decision making, conflict, or problem solving**? Not in all games. Walking simulators are games where the player just explores in an environment, and there are often no puzzles or conflicts at all.

* It's actually pretty hard to pin down exactly what a game is. If you read textbooks on game design, you'll find all sorts of compelling definitions. So for our purposes, let's define the **meaning of "game"** like this:

A game is a program that lets you play with it in a way that (hopefully) is as entertaining to play as it is to make.

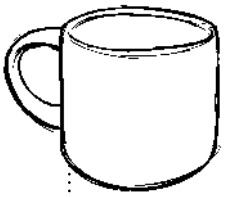
Break up large projects into smaller parts

Our goal in this book is to help you to learn C#, but we also help you **become a great developer**, and one of the most important skills great developers work on is tackling large projects. You'll build a lot of projects throughout this book. They'll be smaller starting with the next chapter, but they'll get bigger as you go further. As the projects get bigger, we'll show you how to break them up into smaller parts that you can work on one after another. This project is no exception—it's a larger project, like the ones you'll do later in the book—so you'll do it in five parts.



The goal of this project is to help get you used to writing C# and using the IDE. If you run into any trouble with this project, you can watch a full video walkthrough on our YouTube channel: *https://www.youtube.com/@headfirstcsharp*

You can download all of the code and a PDF of this chapter from our GitHub page: *https://github.com/head-first-csharp/fifth-edition*



Relax

This chapter is all about learning the basics, getting used to creating projects, editing code, and building your game.

Don't worry if there are things that you don't understand yet. By the end of the book, you'll understand everything that's going on in this game. For now, just follow the step-by-step instructions to get your game up and running. This will give you a solid foundation to build on later.

Here's how you'll build your game

You'll build your animal matching game using **.NET MAUI** (which stands for .NET Multi-platform App UI, or just **MAUI**). MAUI is a technology that you can use to create apps in C# that run natively as desktop apps on Windows and macOS, or as mobile apps on your Android or iOS mobile devices.

The rest of this chapter will walk you through building the game. You'll be doing it in a series of separate parts:

1. **First you'll create a new .NET MAUI project in Visual Studio.**
   You just created a new console application. Now you'll create a new MAUI app.

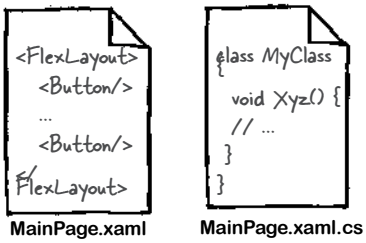
2. **Then you'll use XAML to design the page.**
   Individual screens in MAUI apps are called **pages**. You'll design them using XAML, a design language you'll use to define how those pages work.

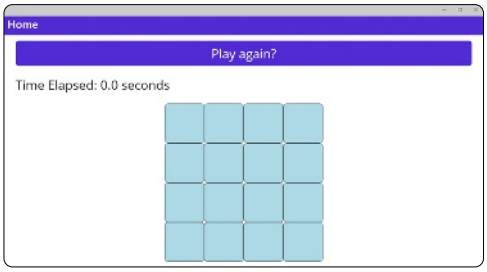
3. **You'll write C# code to add random animal emoji to the page.**
   When your app first loads, it will run that code to display 16 buttons with eight pairs of animal emoji in a random order.

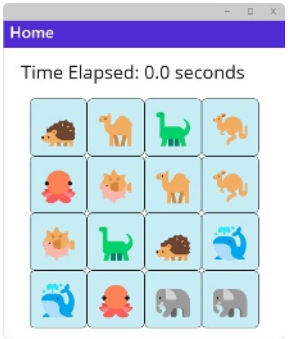
4. **You'll make the gameplay work.**
   The game needs to detect when the user clicks on pairs of emoji, keep track of the pairs, and end the game when they've found all of the matches. You'll write that code too.

5. **Finally, you'll make the game more exciting by adding a timer.**
   Your timer will start when the player starts the game, and keep track of how long it takes the player to find all eight pairs of animals.

This project can take anywhere from 20 minutes to over an hour, depending on how quickly you type. We learn better when we don't feel rushed, so give yourself plenty of time.

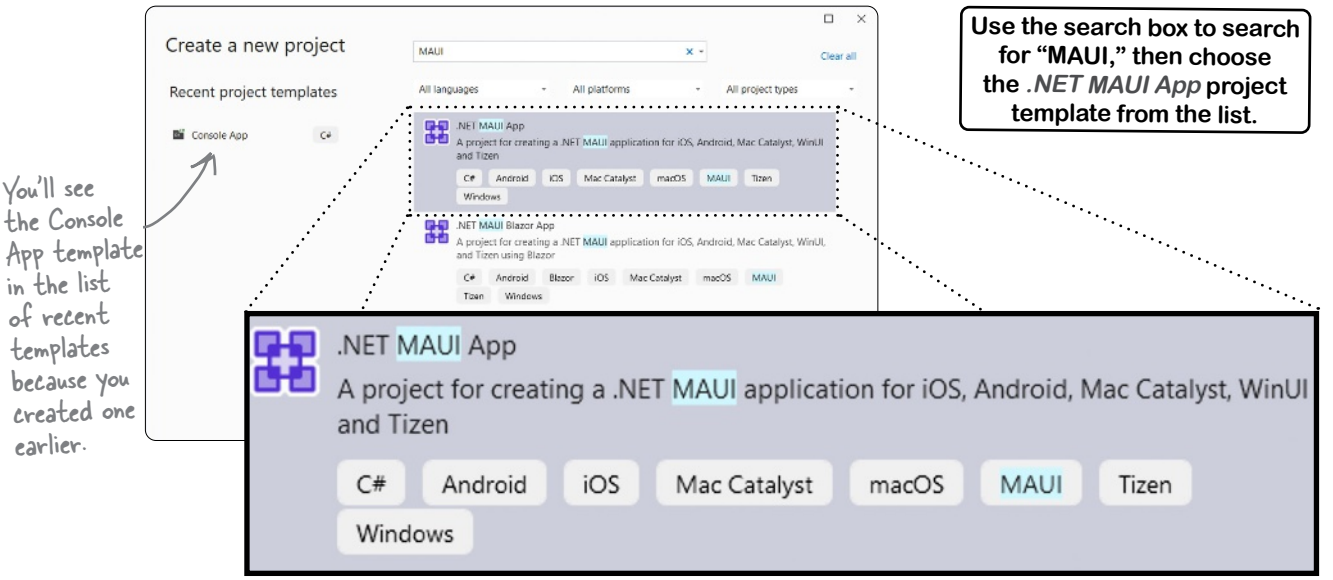






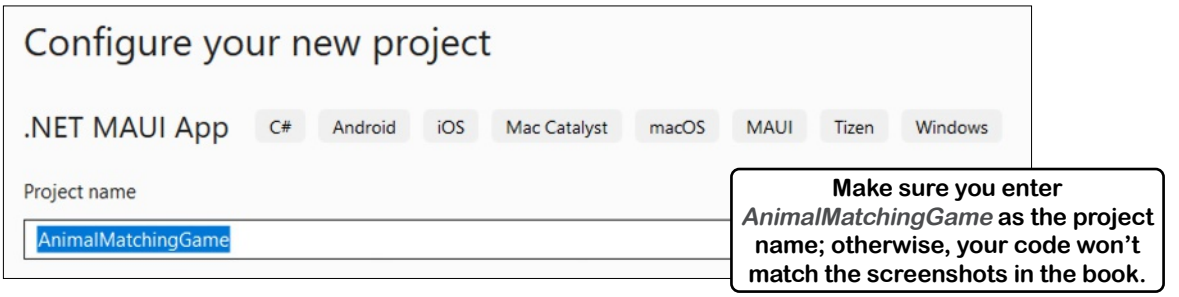
Create a .NET MAUI project in Visual Studio

You can create a .NET MAUI app in Visual Studio just like you did with the console app at the beginning of the chapter, using the "Create a new project" button displayed when you first open Visual Studio. If it's already open, choose New >> Project (Ctrl+Shift+N) from the File menu to bring up the "Create a new project" window.



Choose the **.NET MAUI App project template** and click Next. Visual Studio will prompt you for a project name, just like it did when you created a Console App project.

Enter **AnimalMatchingGame** as the project name and click Next.



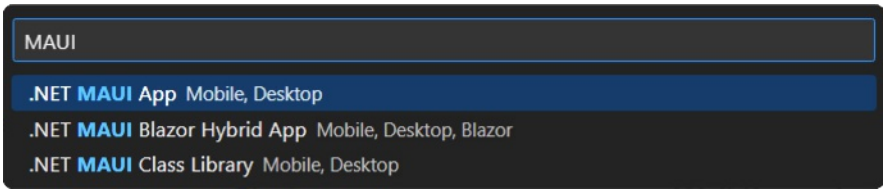
Finally, Visual Studio will ask you to choose a version of .NET—choose the latest version, just like you did when you created the Console App project. Then click the Create button to create your new .NET MAUI project.

Create a .NET MAUI project in Visual Studio Code

If you're using Visual Studio Code, creating a .NET MAUI project is really similar to creating the Console App project, just like you did at the beginning of the chapter. First, **close your current app** by choosing **File >> Close Folder** (Ctrl+K F or ⌘K F). It's *really important to close your folder*; otherwise, you'll add a new project to the same solution.

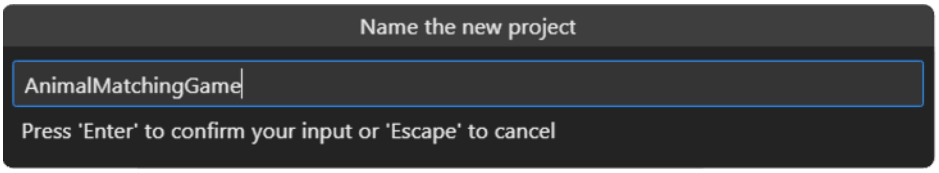
Next, create the .NET MAUI App project. Use Ctrl+Shift+P or ⇧⌘P to **open the Command Palette**. Choose the command **.NET: NEW Project** to create a new project. VSCode will prompt for the project type.

Choose the **.NET MAUI App** project type. You can type "MAUI" to filter the options.



Make sure you enter *AnimalMatchingGame* as the project name; otherwise, your code won't match the screenshots in the book.

VSCode will ask you to give the project a name. **Name your project AnimalMatchingGame**.



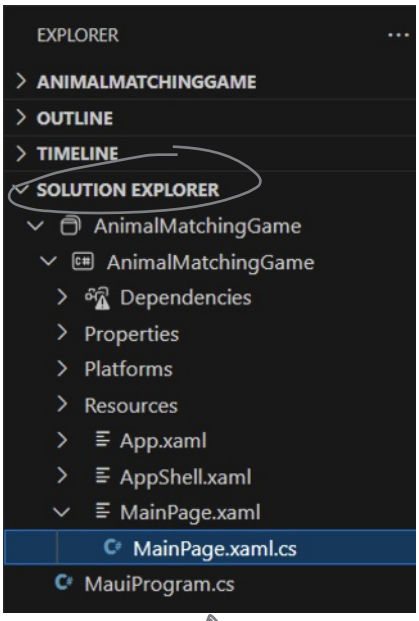
***Expand* the Solution Explorer and *collapse* all of the other sections of the Explorer window.**

VSCode will prompt you for a directory. Choose the **default directory**.

You should now see your project in the Solution Explorer at the bottom of the Explorer panel.

When it comes time to run your project, do the following—this is ***different from running a Console App***:

1. Expand the Solution Explorer in the Explorer panel.
2. Expand the file *MainPage.xaml* to reveal MainPage.xaml.cs (it may be expanded already).
3. Click on MainPage.xaml.cs to select it.
4. Open the Command Palette (Ctrl+Shift+P or ⇧⌘P) and choose **Debug: Start Debugging**. You can also **open MainPage.xaml** and either **press F5** or choose Start Debugging fom the Run menu.
5. VSCode may prompt you to select a debugger. Select **.NET MAUI**. Once you do that, your app should start running in a new window.



The Solution Explorer section may be at the very bottom of the Explorer window.

Run your new .NET MAUI app

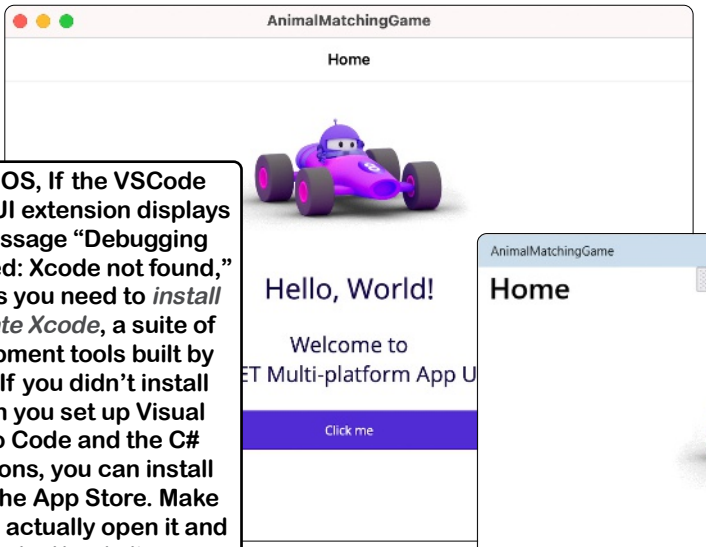
Do this!

In Visual Studio: click Windows Machine in the toolbar or choose **Start Debugging (F5)** from the Debug menu.

In Visual Studio Code: open MainPage.xaml and choose Start Debugging (F5) from the Run menu. If it prompts you to select a debugger, choose .NET MAUI from the list. On macOS you may see a prompt warning that AnimalMatchingGame is from an unidentified developer, and asking if you're sure you want to open it. **Click Open Anyway.**

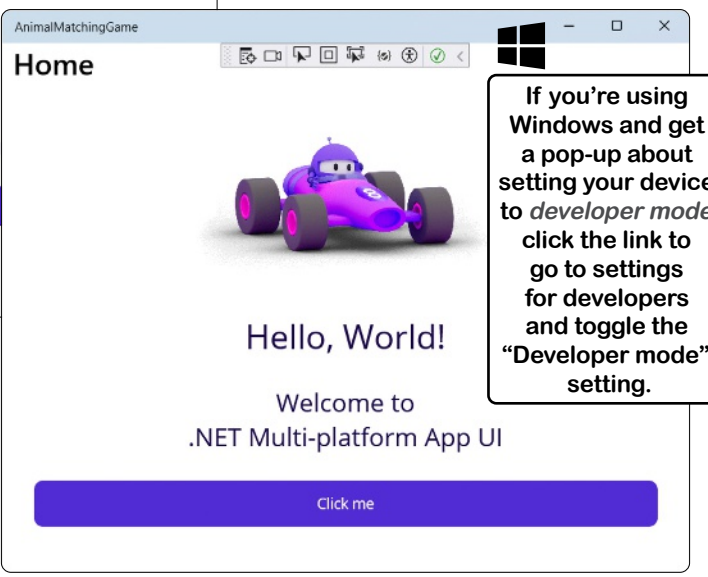
When you see Do this! (or Now do this!, or Debug this!, etc.), go to Visual Studio and follow along. We'll tell you exactly what to do, and point out what to look for to get the most out of the example we show you.

The IDE will ***build*** your code, which means converting it to an executable program that your operating system can run. Then it will start your app:



These screenshots were made with .NET 8. If you're using a different version of .NET, the default MAUI app may look a little different.

In macOS, If the VSCode .NET MAUI extension displays the message "Debugging cancelled: Xcode not found," it means you need to *install or update Xcode*, a suite of development tools built by Apple. If you didn't install it when you set up Visual Studio Code and the C# extensions, you can install it from the App Store. Make sure you actually open it and *accept the Xcode license agreement*; otherwise, you may get an error when you try to debug a MAUI app in Visual Studio Code.



If you're using Windows and get a pop-up about setting your device to *developer mode*, click the link to go to settings for developers and toggle the "Developer mode" setting.

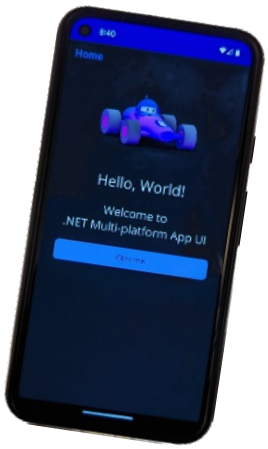
Stop your MAUI app

You can stop your app by **closing the app window**. You can also choose **Stop Debugging (Shift+F5)** from the Debug menu in Visual Studio or Run menu in VSCode, or **click the square Stop** button in the IDE's toolbar.

You can start or stop your app at any time. If there are syntax errors (like typos or invalid keywords) in the C# or XAML code, **the IDE won't be able to run the app**.

For tips on running your app, see *https://github.com/head-first-csharp/fifth-edition*.

MAUI apps work on all of your devices



MAUI is a **cross-platform framework** for building visual apps, which means the apps that you build can run on your Android and iOS devices. Many of the chapters in this book include .NET MAUI projects so you can learn to build more visual apps.

You can run MAUI apps on an Android device directly from Visual Studio. This page shows you how to set up an Android device so you can connect it to your computer and run your MAUI apps on it: *https://learn.microsoft.com/dotnet/maui/android/device/setup*

You can also run MAUI apps on your iOS device, but it requires a little more setup—and it costs money because you need to join the Apple Developer Program. This page walks you through the process: *https://learn.microsoft.com/dotnet/maui/ios/device-provisioning*

MAUI apps are designed with XAML

XAML (the X is pronounced like Z, and it rhymes with "camel") is a **markup language** that you'll use to build the user interfaces for your MAUI apps. XAML is based on XML (so if you've ever worked with HTML, you have a head start). Here's an example of a XAML tag for a button:

```
<Button Text="Click" Clicked="Button_Click" />
```

This book is about learning C#, so we'll give you **just enough** XAML so you can build great-looking MAUI apps—and we'll make sure that you have a solid foundation for learning more.

Are you seeing errors or having problems in Visual Studio Code?

If you see an error window when you try to run your code—it may have a message like "Android SDK: Install required" or a warning about licenses—then there are some installation steps that you still need to do:

https://learn.microsoft.com/dotnet/maui/get-started/installation?tabs=visual-studio-code

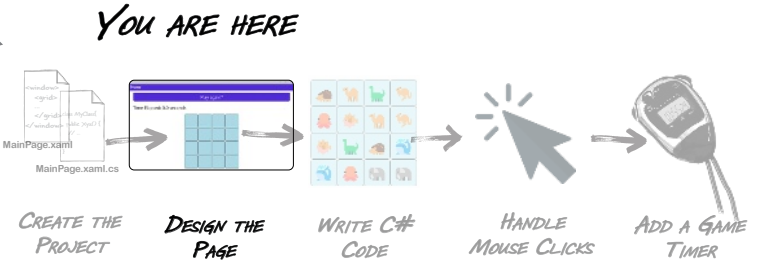
Follow all of the steps on that page. Make sure you've installed all of the VSCode extensions, the .NET and .NET MAUI workloads, the Android SDK, and the latest XCode if you're using a Mac. If you're still running into problems, create a new .NET MAUI project and watch the Terminal window for error messages as soon as the project opens, and look for instructions on how to accept a license. Follow those instructions exactly. You may need to install the most recent version of OpenJDK first: https://learn.microsoft.com/java/openjdk/download

If you keep getting license errors on Windows, open the Command Palette and choose .NET MAUI: Configure Android and then Review Android Licenses to accept the licenses. You may need to do this while running VSCode as administrator. You can also choose "How to Configure Android" to see a web page with complete instructions.

If you're using a Mac, are you getting any of the following errors *when you run your app: "No debug target available, skipping debugging," an error about Android or iOS, or an error about prelaunch tasks terminating? Press ⇧⌘P to open the command palette, choose* ***.NET MAUI: Pick macOS Device****, and select your computer from the list, then* ***open MainPage.xaml*** *and* ***press F5*** *to run your app again.*

If MAUI is still giving you problems, Microsoft has a page to help troubleshoot it: *https://learn.microsoft.com/dotnet/maui/troubleshooting*

We'll include a "map" like this at the start of each of the sections of this project to help you keep track of the big picture.

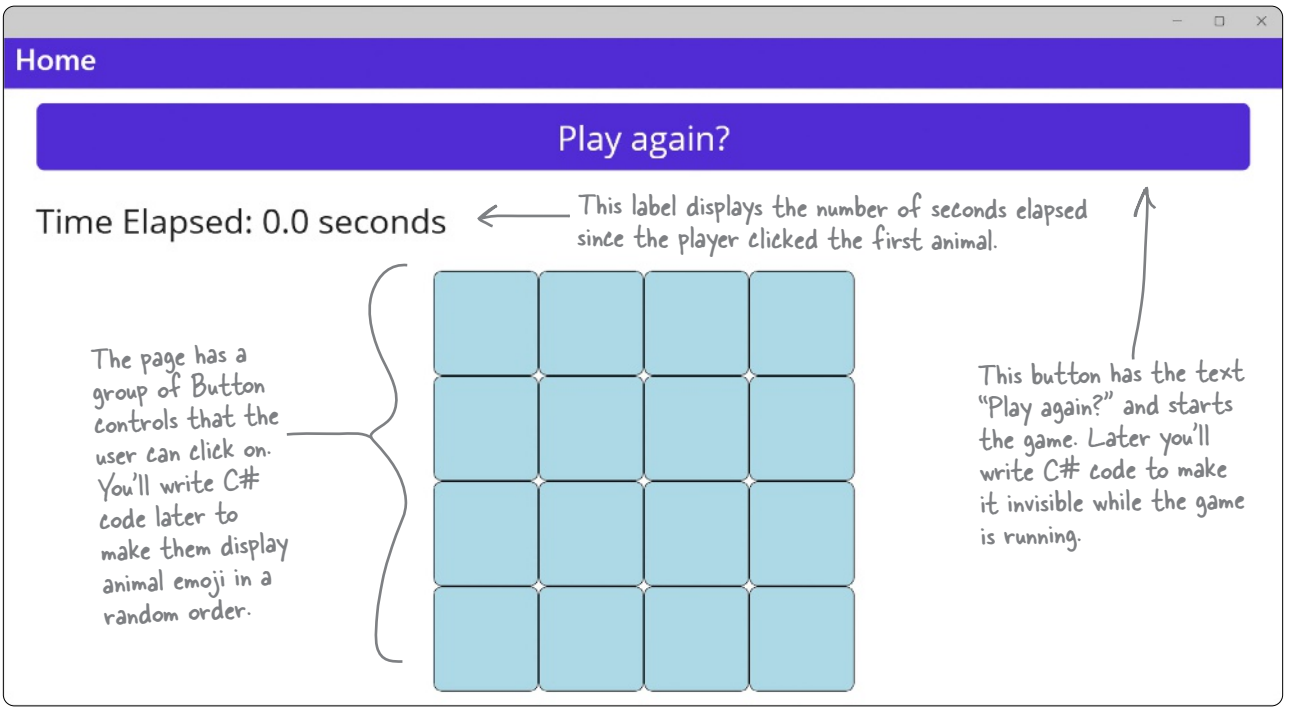


Here's the page that you'll build

When you start a project, the first thing you always want to do is take a few minutes to understand the big picture. What are you going to create? How will it work? Let's take a look at the page you're about to build.

When you open an app built with .NET MAUI, the first thing it shows you is a **page** that you interact with. That page uses **controls**, or visual widgets like buttons and labels, to create a user interface (or UI) that you can interact with. Here's the page that you're going to design:

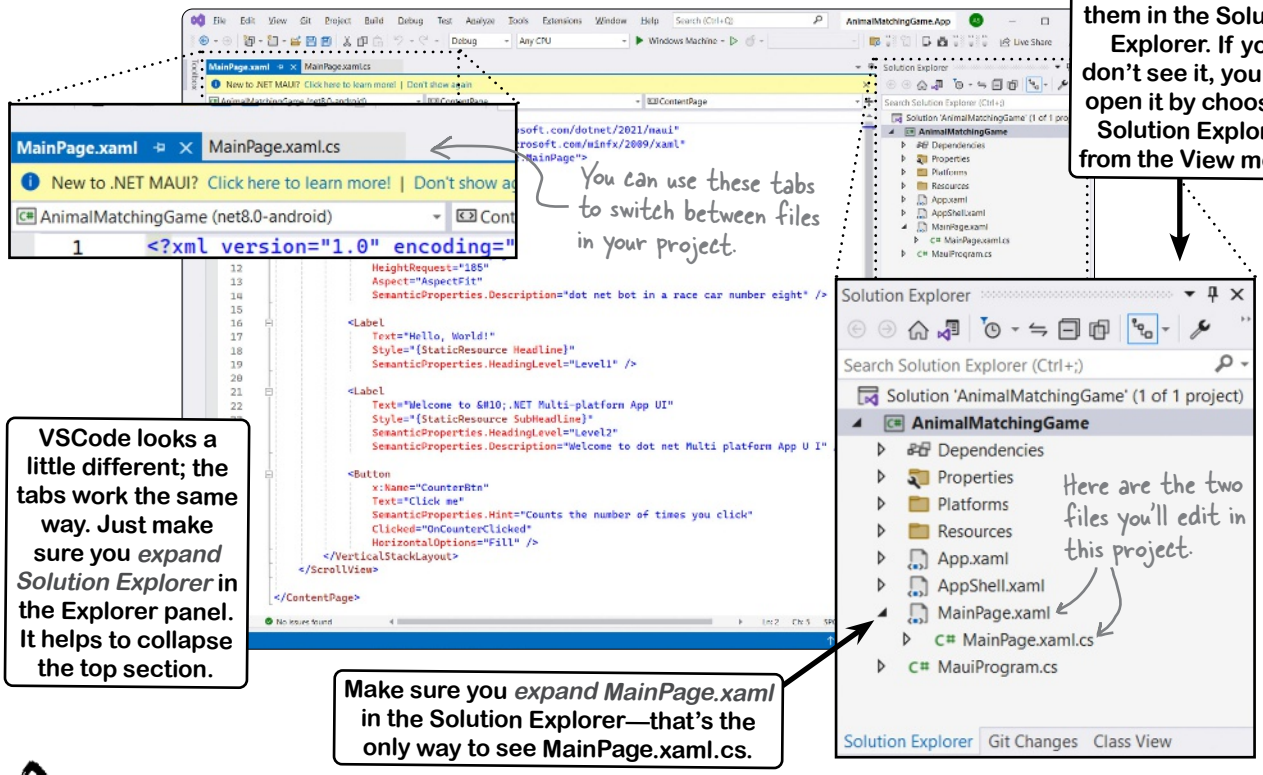
You'll use XAML to lay out your page. A lot of C# developers consider XAML a core skill, and many C# jobs require you to know at least some XAML, so we wanted to make sure to include enough of it in this book to give you a good grounding in it.



Start editing your XAML code

The Solution Explorer lets you edit code files by double-clicking on them (or single-clicking in VSCode). We'll work with two files: *MainPage.xaml* (which contains your XAML code) and *MainPage.xaml.cs* (which has the C# code for your game). This is what it looks like in Visual Studio:

If you close any of the files, you can open them again by double-clicking on them in the Solution Explorer. If you don't see it, you can open it by choosing Solution Explorer from the View menu.



VSCode looks a little different; the tabs work the same way. Just make sure you *expand Solution Explorer* in the Explorer panel. It helps to collapse the top section.

Make sure you *expand MainPage.xaml* in the Solution Explorer—that's the only way to see MainPage.xaml.cs.

Watch it!

Visual Studio Code may have limited support for editing XAML code.

When you use Visual Studio Code to edit the XAML code in MainPage.xaml, you may notice that it's missing some of the features that we talk about in this chapter. When you edit XAML code in Visual Studio, it has these really convenient features:

- *Visual Studio has a Toolbox with controls that you can drag into the XAML editor, or a Properties window that makes it easy to edit their properties.*
- *When you add an event, Visual Studio automatically creates the code for the event handler:*

```
FontSize="Large" Clicked="" />
                          <New Event Handler>
```

Visual Studio helps you connect your event to an event handler or create a new one.

At the time we're writing this book, the .NET MAUI extension for VSCode does not include these features, so you'll have to write code manually instead of letting the IDE generate it for you.

Add the XAML for a Button and a Label

The first thing we'll do is design the page for the game. It will have 16 buttons to display the animal emoji, plus a "Play again?" button to restart the game when the player wins.

Do this!

1 Delete everything between the opening and closing VerticalStackLayout tags.

XAML is a **tag-based markup language**. That means your XAML code uses **tags** to define everything that appears in your app. Here's an example of a tag—you can find it near the top of *MainPage.xaml*:

```
<ScrollView>
```

That's an **opening tag**. You can find its matching **closing tag** near the end of the file: `</ScrollView>`.

These tags add a **ScrollView control** to the page. If your app is in a window that's smaller than its contents, everything between the opening and closing tag can be scrolled up and down.

Find the opening VerticalStackLayout tag. It's on the next few lines of the file, and it looks like this:

```
<VerticalStackLayout
     Padding="30,0"
     Spacing="25">
```

If you're using a different version of .NET than we did, the XAML in the page may start out a little different. That's OK—you just need to get the XAML for your page to *exactly match* the XAML code below.

Next, find the closing VerticalStackLayout tag:

```
</VerticalStackLayout>
```

Now **carefully delete all of the lines between those two tags**. The XAML code in your *MainPage.xaml* file should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">

    <ScrollView>

       <VerticalStackLayout
            Padding="30,0"
            Spacing="25">



        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

In the next step, you'll put your new XAML code right here, where you deleted the old code.

At the time we're writing this, the .NET MAUI extension for Visual Studio Code *does not have a Toolbox*. If you're using Visual Studio Code, you may not be able to do all of the fancy drag-and-drop XAML editing you get with Visual Studio, so you'll need to carefully type in all of the XAML line by line. But don't worry, your app will still work the same way.

② Delete the C# code that goes with the XAML that you just deleted.

If you try to run your app right now, Visual Studio will give you an error message and refuse to run it, because the C# code depends on things you just deleted. **Expand MainPage.xaml in the Solution Explorer and open *MainPage.xaml.cs*** and find this code:

```
private void OnCounterClicked(object sender, EventArgs e)
```

Delete it, and the next 10 lines of code, up to and including the closing curly brace `}`. Be careful not to delete the final closing `}` at the end of the file. Then delete this line of code: `int count = 0;`

Your C# code should now look like this:

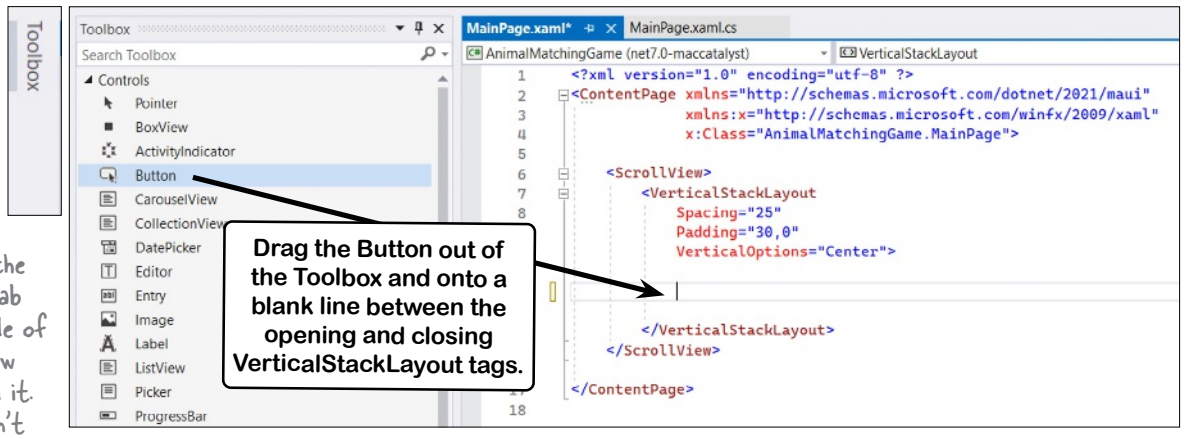
```
namespace AnimalMatchingGame;

public partial class MainPage : ContentPage
{
        public MainPage()
        {
                InitializeComponent();
        }
}
```

You'll learn about classes and namespaces in Chapter 3.

Make sure you go back to the Solution Explorer and open MainPage.xaml.cs to edit the C# code. You may need to expand MainPage.xaml to see it.

③ Go back to MainPage.xaml and use the Toolbox to add the "Play Again?" button.

You'll be editing the XAML code again, so **switch back to the MainPage.xaml tab**. If you don't see the Toolbox panel, expand it by clicking the tab on the side of the window. **Add a few extra blank lines** where you deleted the code between the opening and closing VerticalStackLayout tags. Then **drag the Button out of the Toolbox** and drop it onto one of the lines that you added.



Click on the Toolbox tab on the side of the window to expand it. If you don't see it, choose Toolbox from the View menu. Click the pin button in the upper-right corner to keep it from collapsing.

You should now see a new Button tag between the VerticalStackLayout tags—it's OK if the spacing or indenting is a little different, because extra spaces or lines don't matter in XAML:

```
<VerticalStackLayout
    Padding="30,0"
    Spacing="25">

  <Button Text="" />

</VerticalStackLayout>
```

When you dragged the Button out of the toolbox and into your code, Visual Studio added this Button tag. If you're using VSCode, you may not have a Toolbox, so type this in exactly like it appears here.

If you're using Visual Studio Code, you may not have a Toolbox panel or Properties window. You'll need to type the XAML into the MainPage.xaml file so it exactly matches our code.

4 Add properties to the XAML tag for the "Play again?" button.

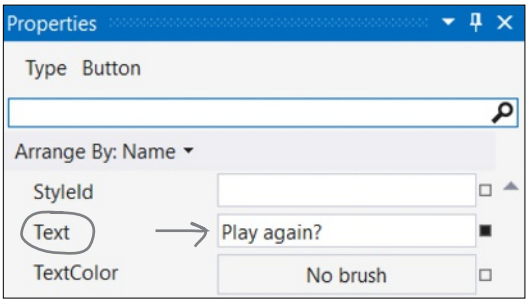
XAML tags have **properties** that let you set options to customize how they're displayed on the page. The **Properties window** in Visual Studio makes it easier to edit them.

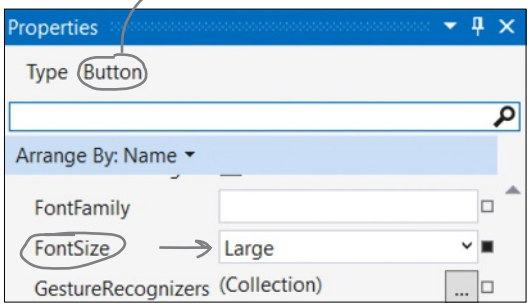
Click on the code for the Button tag in *MainPage.xaml*, so your cursor is somewhere between the opening **<** and closing **>** angle brackets. Then look at the **Properties window**—it's usually docked in the lower-right corner of Visual Studio. If you don't see it, choose Properties or Properties Window from the View menu. Make sure it says "Type Button" at the top, so you know that you're editing the button.

Find the ***Text*** property and **set it to "Play again?"**

Then find the ***FontSize*** property and **set it to "Large."**

Make sure you're editing the Button.





When you're done editing the button, the XAML for it should look like this:

```
<Button Text="Play again?" FontSize="Large" />
```

The Button tag now has **Text and FontSize properties**.

5 Edit the XAML code by hand for your button to give it a name.

You can also edit XAML code by hand—for example, if you run into trouble with the Properties window, you could type the XAML directly into the editor. ***You need to make sure that you copy all of the brackets, quotes, etc. exactly, otherwise your code won't run!***

In the next part of the project, you'll write C# code to make your "Play again?" button visible when the game is over, and invisible while the game is running. You'll give it a **name** that the C# code can use to tell it to show or hide itself.

Use the editor to **add an x:Name property** to give your button a name. It should look like this:

```
<Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large" />
```

XAML tags have properties that let you set options to customize how they're displayed on the page.

6 Add an event handler so your button does something.

When you click a button, it executes C# code called an **event handler**. Visual Studio makes it easy to add one. Place your mouse cursor just before the **/>** at the end of the Button tag and start typing **`Clicked`**. Visual Studio will pop up an IntelliSense window:

```
FontSize="Large" Cli />
```

Choose Clicked from the list and either click on it or press Enter. Visual Studio will then show you this:

```
FontSize="Large" Clicked="" />
```

Press Enter to add a new event handler. Your XAML tag should now look like this:

A XAML tag can be on a single line or split across multiple lines. Make sure you put the break in a space (not the space in "Play again").

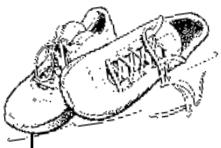
```
<Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
        Clicked="PlayAgainButton_Clicked" />
```

Switch to the MainPage.xaml.cs tab. You can see the code that Visual Studio added, which looks like this:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{

}
```

If you're using Visual Studio Code, you might not get the <New Event Handler> pop-up, so you'll need to add this code to MainPage.xaml.cs by hand. Your app will still work just fine.

When you see an exercise, that's your chance to get some practice on your own. Make sure you do every exercise—they're an important part of the book. If an exercise is part of a project, then the project won't work until you get it right. But don't worry—we'll always give you the solution. And if you get stuck, it's always OK to peek at the solution!



Exercise

Add a Label control to your XAML page.

Go back to the screenshot of the game that shows the "Play again?" button. Notice how it also has text above the button that displays the time elapsed? That's a Label. It's up to you to add a tag for it. Here's what you'll do:

1. Switch to the MainPage.xaml tab.
2. Open the Toolbox and **drag a Label** into your XAML code. Make sure it gets added directly below the Button, just like you did in Step 3 when you were adding the Button.
3. Use the Properties window to set the **Text button to "Time Elapsed: 0.0 seconds"** and the **FontSize to "Large"** just like you did in Step 4 when you were adding the Button.
4. Edit the XAML code by hand and **set the x:Name to "TimeElapsed"** just like you did in Step 5 when you were adding the Button.

If you're using VSCode and don't have a Toolbox, skip straight to the Exercise Solution and carefully add the `<Label ... />` tag.

Exercise Solution

Add a Label control to your XAML page.

If you followed the steps in the exercise correctly, your XAML code in *MainPage.xaml* should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">

    <ScrollView>
       <VerticalStackLayout
            Padding="30,0"
            Spacing="25">

            <Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
                    Clicked="PlayAgainButton_Clicked" />

            <Label x:Name="TimeElapsed" Text="Time Elapsed: 0.0 seconds"
                   FontSize="Large" />

       </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

It's OK if there are line breaks *between* the properties in a tag, or if the properties are in a different order.

Here's the Label that you added in the exercise. We added a line break between the Text and FontSize properties, but you don't have to.

The C# code in *MainPage.xaml.cs* didn't get modified as part of the exercise, so it should still look like this—if you're using Visual Studio Code, you had to carefully type in the PlayAgainButton_Clicked method by hand:

```
namespace AnimalMatchingGame;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PlayAgainButton_Clicked(object sender, EventArgs e)
    {

    }
}
```

Before you move on, make sure that your XAML and C# code matches ours *exactly*, and that your app looks like our screenshot when you run it.

If you run your app now, it should look like this.

Press F5 to run your app again. In VSCode you need to switch back to the XAML code in MainPage.xaml and then press F5.

there are no Dumb Questions

Q: What exactly is a "page" in a MAUI app?

A: A .NET MAUI app is usually built out of one or more **pages**, or individual screens that have different layouts and contain **controls** like labels and buttons. Some MAUI apps have multiple pages that let you navigate between them. Your Animal Matching Game app will just have a single page with 16 animal buttons, a "Play again?" button, and a label to show the elapsed time.

Q: So those buttons and labels are controls?

A: Yes. Everything you see on a MAUI page is a control—including the page itself, which is a ContentPage control. Some controls are dedicated to making your page look a certain way, like the VerticalStackLayout control that causes other controls to be stacked one on top of another. Others, like the Button and Label controls, are there to display some kind of widget that the user can see and interact with. We'll talk more about controls in the next chapter.

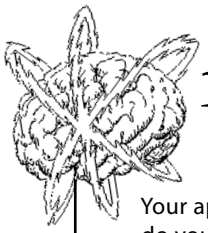
Q: It looks like some controls contain others, like the VerticalStackLayout in my app contains Button and a Label. What's going on there?

A: When you include layout control like VerticalStackLayout on your page, you can't actually see it. Its whole purpose is to cause the other controls on the page to be displayed a certain way—in this case, to be stacked on top of each other. You need a way to tell MAUI which other controls on the page you want it to stack. To do that, you **nest** those other controls inside the VerticalStackLayout by including their tags between its opening <VerticalStackLayout> tag and its closing </VerticalStackLayout> tag.

Q: Why do some tags like `<ScrollView>` have a closing `</ScrollView>` tag, but others like `<Button>` don't have one?

A: A Button control doesn't need to have any other controls nested inside of it, so there's no need for it to have a closing tag—instead, you can just end the tag with **/>** to make it **self-closing**.

These Brain Power boxes are here to give you something to think about. When you see one, don't just go on to the next section. Take a few minutes and actually think about what you're being asked. That will really help you get this material into your brain faster!



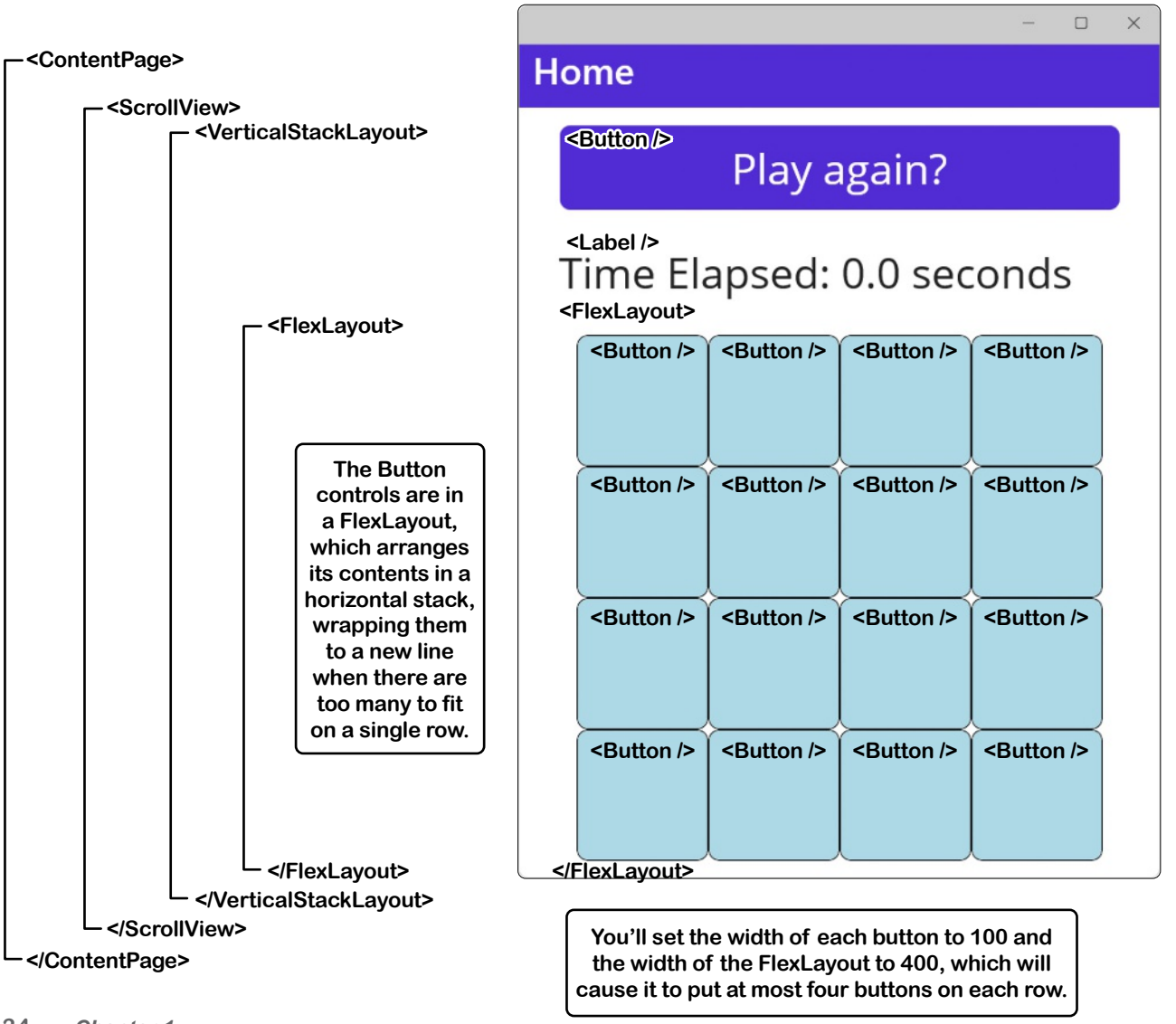
Brain Power

Your app is looking good so far, but now you need to add some buttons. How do you think you'll do that? What do you think you'll have to add to the XAML to get 16 buttons to be displayed in a layout with four rows of four buttons?

Use a FlexLayout to make a grid of animal buttons

The XAML for your page currently has three tags that determine its layout: there's a ContentPage tag on the outside that displays the whole view. It contains a ScrollView—everything nested between its start and end tags will scroll if it goes off the bottom of the page. Inside it is a VerticalStackLayout, which causes everything between its start and end tags to be stacked on top of each other in the order that they appear. Inside all of those tags are self-closing Button and Label tags.

The next thing you'll do is add a **FlexLayout**, which arranges anything inside of it in rows, wrapping them to the next row so they all fit inside its total width. You'll add 16 Button tags inside the FlexLayout. You'll get them to display in a 4x4 grid by setting the width of each button to 100 and the width of the FlexLayout to 400, so exactly four buttons will fit on each row.



Exercise

This looks like a big exercise! But don't worry, just take it step by step. We know you can do it! And remember, it's *not cheating* to look at the solution…in fact, seeing the solution is a great way to help you learn.

It's time to finish designing your page. In this exercise, you'll add a FlexLayout underneath the Label that you added in the last exercise. Next, you'll set its properties. Then you'll add a button. And finally, you'll copy the XAML for that button and paste it 15 more times, so you have a total of 16 buttons on your page. If you're using VSCode and don't have a Toolbox, just type the XAML exactly how it appears in the instructions instead of dragging it out of the Toolbox.

Add extra space for your FlexLayout control

Take a careful look at the screenshot that we just showed you. It shows you how the whole page works. Now go back to Visual Studio and look at the XAML for your page, and figure out exactly where the FlexLayout should go—just below the `<Label ... />` tag.

Now put your cursor at that location and press Enter a few times to give yourself space to drag the FlexLayout.

Add the FlexLayout control just below the Label

1. Open the Toolbox and **drag a FlexLayout** into your XAML code. Make sure it gets added directly below the Label, into the extra space you just added. It will look like this: `<FlexLayout></FlexLayout>.`
2. Position your cursor between the **>** and **<** in the middle of the XAML you just added and **add several blank lines** between the opening and closing tags (you'll drag a button into that space later in the exercise).
3. Place your cursor directly on the opening `<FlexLayout>` tag. Make sure the Properties window shows that the type is FlexLayout.
4. Use the Properties window to set the **Wrap property** to "**Wrap"** and the **MaximumWidthRequest property** to "**400.**"
5. Edit the XAML code by hand and **set the x:Name to "AnimalButtons"** just like you did in the last exercise.

Add the first Button inside the FlexLayout

1. Open the Toolbox and **drag a Button** into your XAML code. Make sure it gets added in the space that you added between the opening and closing tags of the FlexLayout. It will look like this: `<Button Text="" />`.
2. Place your cursor inside the `Button` tag. Make sure the Properties window shows that the type is Button.
3. Use the Properties window to set the Button's **HeightRequest property** to "**100**," the **WidthRequest property** to "**100**," and the **FontSize property** to "**60**." The dropdown in the Properties window won't have numbers—you can either type "60" into the window or choose Caption from the dropdown list to set the font size.
4. Edit the XAML for the button and **delete the Text property** by selecting it in the code editor and pressing Delete. Your cursor should now be inside the Button control.
5. Keep the cursor where it is and edit the XAML code by hand to **set the BackgroundColor property** to "**LightBlue**," **the BorderColor property** to "**Black**," and **the BorderWidth property** to "**1**." Visual Studio's IntelliSense pop-up will help you match the colors (but if you're using Visual Studio Code, you may not get a nice pop-up).
6. Add a **Clicked event handler**, just like you did with PlayAgainButton. **Choose <New Event Handler>** from the dropdown, so it creates a new event handler method in the C# code. Use the default name Button_Clicked.

If you're using VSCode, you might need to enter it by hand into MainPage.xaml.cs. It will be just like PlayAgainButton_Clicked, except without "PlayAgain" in the name.

Add the rest of the Buttons

Copy the `<Button ... />` tag that you just added. Then **paste 15 identical tags below it**. You should now have a total of 16 identical Button tags inside a FlexLayout just below the Label. Run your app—it should match our screenshot.

Exercise Solution

It's time to finish designing your page. In this exercise, you'll add a FlexLayout underneath the Label that you added in the last exercise. Next, you'll set its properties. Then you'll add a button. And finally, you'll copy the XAML for that button and paste it 15 more times, so you have a total of 16 buttons on your page.

If you followed the steps in the exercise correctly, your XAML code in *MainPage.xaml* should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">
```

If you chose a different name for your project, you'll see that name here.

```
  <ScrollView>
    <VerticalStackLayout
            Padding="30,0"
            Spacing="25">
```

This looks like a lot of XAML code, but most of it is just the 16 identical Button tags that you copied and pasted.

```
    <Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
                   Clicked="PlayAgainButton_Clicked" />

    <Label x:Name="TimeElapsed" Text="Time Elapsed: 0.0 seconds" FontSize="Large" />

    <FlexLayout x:Name="AnimalButtons" Wrap="Wrap" MaximumWidthRequest="400">

     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
      <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
        HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
     <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
```

Make sure the opening tag of your FlexLayout is right below the Label, and that its properties match ours. Be careful to add a Maximum and not Minimum width request or your buttons won't be in a 4x4 grid.

It looks like there is a lot of XAML code here! But most of the XAML that you added is the same `<Button ... />` tag *copied and pasted 16 times*.

Exercise Solution

The 16 Button tags should be identical. It's OK if the properties are in a different order.

```
      HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
    <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
      HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
    <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
      HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
    <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
      HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
    <Button BackgroundColor="LightBlue" BorderColor="Black" BorderWidth="1"
      HeightRequest="100" WidthRequest="100" FontSize="60" Clicked="Button_Clicked"/>
    </FlexLayout>

  </VerticalStackLayout>
 </ScrollView>

</ContentPage>
```

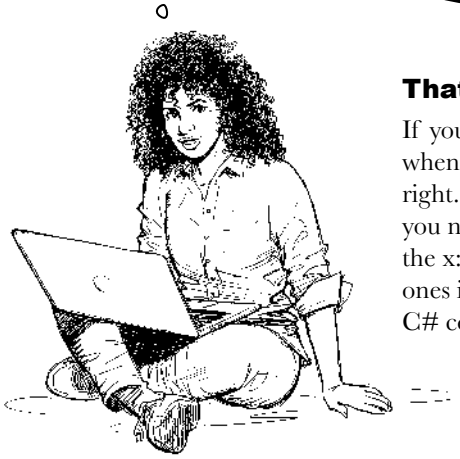
Make sure every button has the `Clicked="Button_Clicked"` property. If the Clicked event handler has a different name, your C# code won't match ours. You can delete the Clicked property from all of the buttons, then *re-add it with the correct name*. Once you add the event handler, it will show up in the dropdown when you change the other buttons.

Here's the Button_Clicked event handler method in your *MainPage.xaml.cs* file:

```
private void Button_Clicked(object sender, EventArgs e)
{

}
```

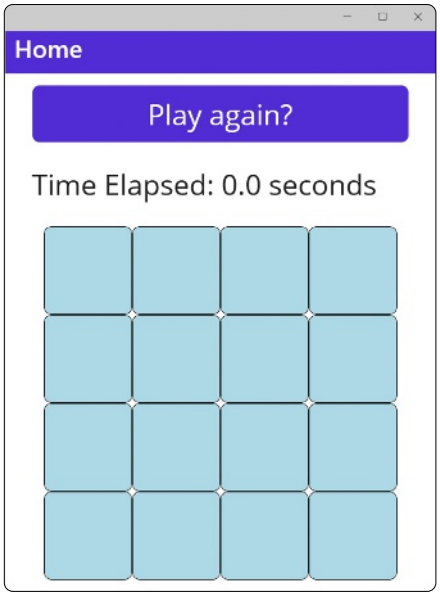
If you're using VSCode, you'll need to open MainPage.xaml.cs and add this event handler method by hand. Make sure it matches exactly!

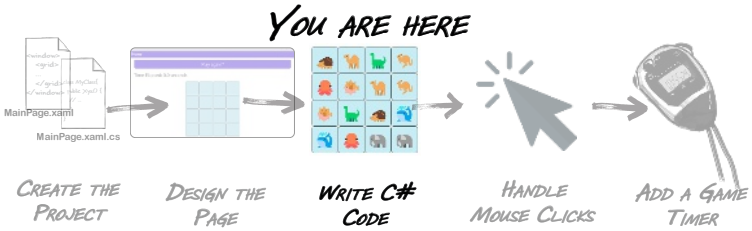
I can check if my solution is right by ***comparing it with the screenshot****. That makes it easier!*



That's right!

If your app looks like this screenshot when you run it, you got things right. There's just one more thing you need to check: make sure that the x:Name properties match the ones in our solution ***exactly***—the C# code you write will use them.





Write C# code to add the animals to the buttons

You started this book to learn C#. You've done all the preparation: creating the project, and designing the page for your app. Now it's time to **get started writing C# code**.

We'll give you all of the code for this project, and show you exactly where it goes. But the goal is to ***get you started learning C#***, so we'll also work with you to help you understand how it all works—and that will provide you with a solid foundation to start writing code on your own.

You'll add code that's run every time the "Play again?" button is clicked. Here's what it will do:

```
Make the animal buttons visible
                ↓
Make the "Play again?" button invisible
                ↓
Create a list of 16 pairs of animal emoji
                ↓
For each of the 16 buttons:

    Pick a random animal from the list
                ↓
    Add that random animal to the button
                ↓
    Remove the animal from the list
                ↓
    Keep going until it runs out of buttons
```

Start editing the PlayAgainButton event handler method

When you were writing the XAML code for the "Play again?" button, you added an event handler:

```
FontSize="Large" Clicked="" />
```

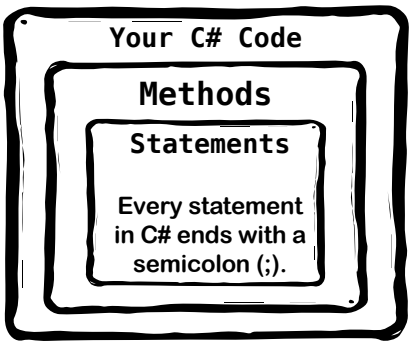
<New Event Handler>

When you did this, Visual Studio added `Clicked="PlayAgainButton_Clicked"` to the XAML tag for the button. It also added this C# code to *MainPage.xaml.cs*:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{

}
```

That's a **method**. C# code is made up of statements, or specific tasks that you're telling your app to execute. Those statements are bundled into methods. Methods have a name—this method is named `PlayAgainButton_Clicked`.

Visual Studio generated that method for you automatically when you added the Clicked event handler to your XAML code to give you a place to add the statements that will tell it what to do when the "Play again?" button is clicked.



Add a C# statement to the event handler method

Place your cursor on the line between the opening `{` curly bracket and closing `}` curly bracket of the method. Then start typing the following line of code to make the animal buttons visible:

```
AnimalButtons.IsVisible = true;
```

Do this!

As you're typing, you'll see some of Visual Studio's really powerful tools that help you write code:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = false;
}
```

Tab Tab to accept

IsLoaded
IsReadOnly
IsSet
IsVisible
AlignItems
AnimationIsRunning
AutomationId
CascadeInputTransparent
ClassId

This is an *IntelliSense* window. Visual Studio displays it as soon as you press the period key. It shows you options that you can use to add to the code you just wrote. If you choose IsVisible, it will auto-complete that part of the statement for you.

When you're typing code, you may see Visual Studio giving you suggestions to complete the entire statement. This is a really powerful feature called *IntelliCode*. It uses an artificial intelligence system trained on millions of lines of code to give you suggestions—and more often than not, it seems to be able to read your mind!

VSCode may not show you suggestions like this. It's a feature of Visual Studio.

Add more statements to your event handler

When the player clicks the "Play again?" button, the app will display the animal buttons, hide the "Play again?" button, and then fill the animal buttons with eight pairs of animal emoji in a random order. You're going to add statements to the PlayAgainButton_Clicked event handler method to do all that.

Do this!

1 **Add a statement to make the "Play again?" button invisible.**

Do you remember how you used the x:Name property in your XAML code to give names to the "Play again?" button and the FlexLayout that contains the 16 animal buttons? Take a minute and go back to that XAML code—you gave the FlexLayout the name "AnimalButtons," and you just added a line of code that used that name.

You also used an x:Name to give the "Play again?" button the name "PlayAgainButton." Now add a second line of code to your event handler method:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = true;
    PlayAgainButton.IsVisible = false;
}
```

Add this line of code right below the one you just added.

That statement turns the "Play again?" button invisible.

We made the code that's already in your files a lighter color to make it easier for you to see what to add.

2 **Make the animal buttons invisible when the app starts.**

Take a closer look at the first statement that you added to your event handler method. It makes the FlexLayout that contains the animal buttons visible. But wait a minute—it's already visible! You saw it when you ran your app. Let's do something about that.

Go back to the XAML code in *MainPage.xaml* and set the IsVisible property to `"false"`:

```
<FlexLayout x:Name="AnimalButtons" Wrap="Wrap"
            MaximumWidthRequest="400" IsVisible="false">
```

Did you notice that you're ***setting the same IsVisible property*** in both the C# code and XAML? When the app starts, the `IsVisible="false"` in the XAML causes the page to display without the FlexLayout and its buttons. When you click the "Play Again?" button, the first line of code in its Clicked event handler method sets that property to true, causing the FlexLayout and its buttons to appear on the page.

Now your app will make the animal buttons invisible when it starts up. As soon as the player clicks the "Play again?" button to start the game, it will show the animal buttons and hide the "Play again?" button.

The properties on controls can be set in both XAML and C# code.

3 Run your app and make sure it works so far.

When you're writing code, you don't just write a complete app from beginning to end, and then run it to see if it works. That's not how it works at all! ***Writing code is a creative process.*** There are many, many ways to make your code do a specific thing, and in a lot of cases, the only way you can really be sure you're happy with it is to try writing it one way—and if you don't like it, change it.

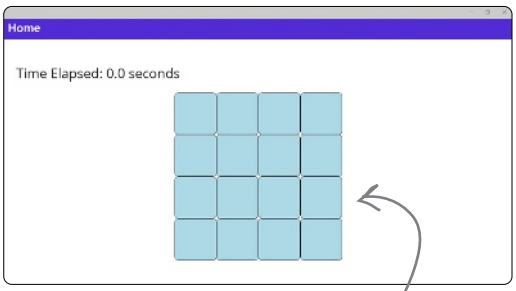
Plus, it's easy to make **syntax errors** in your code. A syntax error means that you wrote something that isn't valid C# code, like using a keyword or symbol incorrectly or using a name that doesn't exist. For example, if you enter an extra **}** closing curly brace at the end of a method and then try to run it, Visual Studio will give you an error telling you that it can't **build** your code (which is what it does to turn your C# code into something that your computer can actually execute).

What does all that mean?

It means that you'll ***run your apps all the time, over and over again***. And that's perfectly fine! It's absolutely OK to run your app after even a tiny change, just to see what that change did. The more comfortable you are running your app, the more you'll feel like you can experiment and make changes—and the more fun you'll have with it.

So go ahead and **run your app now**. Make sure it starts out with the "Play again?" button visible and the animal buttons invisible. Click the "Play again?" button and make sure it hides itself and shows the animal buttons. When you're done, close the app (or stop it from inside Visual Studio).

When you start the app, you'll see the "Play again?" button but no animal buttons.



Click the "Play again?" button to cause it to disappear and the animal buttons to appear.

Watch it!

When you enter your C# code, even tiny errors can make a big difference.

Some people say that you truly become a developer after the first time you've spent hours tracking down a misplaced period. Case matters: `AnimalButtons` is different from `animalButtons`. Extra commas, semicolons, parentheses, etc. can break your code—or, worse, change your code so that it still builds but does something different than what you want it to do. The IDE's ***AI-assisted IntelliSense and IntelliCode features*** *can help you avoid those problems…but it can't do everything for you. It's up to you to make sure your code is right—and that it does what* ***you*** *expect it to do.*

Add animals to your buttons

This game won't be much fun without animals to click on. Let's update the "Play again?" button's event handler method to set up the buttons with eight pairs of emoji positioned randomly on the buttons.

1 **Start creating a List of animal emoji.**

Your event handler method needs to start with eight pairs of emoji, so you're going to write a statement that creates them and stores them in something called a List (you'll learn a lot more about that in Chapter 8).

Switch back to MainPage.xaml.cs and start typing this line of code right *after* the statements that you just added—but ***don't*** end it with a semicolon, because that's not the end of the statement yet:

```
List<string> animalEmoji = [
```

While you're typing, you'll see IntelliSense windows pop up to help you enter that code. The text that you type will be in a bold color, followed by a ***suggestion*** generated by IntelliCode:

```
0 references
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = true;
    PlayAgainButton.IsVisible = false;

    List<string> list = new List<string>();
}
```

We took this screenshot just after we typed the word "List." The rest of the line is in a lighter color to indicate that it's an IntelliCode suggestion.

```
List<>
List<>                                  System.Collections.Generic
ListProxyChangedEventArgs
ListStringTypeConverter
ListView
ListViewCachingStrategy
ListViewSelectionMode
ListBindableAttribute                   System.ComponentModel
ListChangedEventArgs                    System.ComponentModel
```

The IntelliSense pop-up shows different valid words that complete what you started typing. It looks a little different in VSCode, but has the same options.

As soon as you typed the opening square bracket], Visual Studio added a matching one, placing your mouse cursor between the two brackets.:

```
List<string> animalEmoji = [|]
```

← The mouse cursor should now be between the [brackets].

Press Enter, then add a semicolon to the end. Your PlayAgainButton_Clicked method should now look like this:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = true;
    PlayAgainButton.IsVisible = false;

    List<string> animalEmoji = [

    ];
```

← Make sure you add the semicolon after the closing] bracket.

Some people think the plural emoji is emoji, others think it's emojis. We went with emoji—but both ways are fine!

2 **Add a pair of animal emoji to your list.**

Your C# statement isn't done yet. Make sure your cursor is placed on the blank line you added between the brackets. Now let's add **eight pairs of animal emoji**. You can find emoji by going to your favorite emoji website (for example, *https://emojipedia.org/nature)* and copying individual emoji characters. Alternately…

If you're using Windows, use the **Windows emoji panel** (press Windows logo key + period). If you're using a Mac, use the **Character Viewer panel** (press the fn key, or Ctrl+⌘+Space on older Macs).

Go back to your code and add a double quote **"** then paste the character—we used an octopus—followed by another **"** and a comma, a space, another **"**, the same character again, and one more **"** and comma. You might notice Visual Studio helping you enter this list—for example, when you enter a double quote, it adds the closing quote.

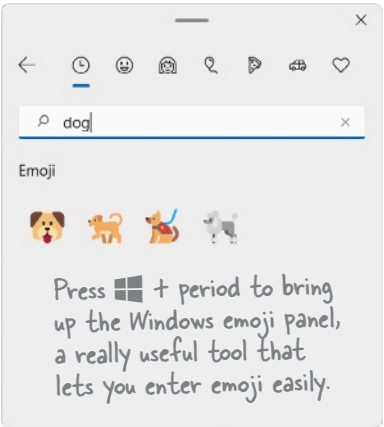
Here's what your list should look like now:

```
List<string> animalEmoji = [
    "🐙", "🐙",
];
```

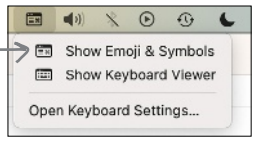
How to enter emoji

If you're using Windows, use the **emoji panel** by pressing Windows logo key ⊞ + period. Use the search box to search for a specific animal. When you find the emoji you want to enter, click on it to enter it as if you'd typed it.

If you're using a Mac, use the **Character Viewer panel**, by pressing Ctrl + ⌘ + space. Use the search box to search for a specific animal. When you find the emoji you want to enter, click on it to enter it as if you'd typed it.



Press ⊞ + period to bring up the Windows emoji panel, a really useful tool that lets you enter emoji easily.



You can also bring up the macOS Character Viewer using the Input menu in the menu bar. If you don't see the Input menu, open System Settings and search for "input menu"—there's an option that you can turn on to show the input menu in the menu bar.

3 Add the rest of the animal emoji pairs to your list.

Then do the same thing for seven more emoji so you end up with **eight pairs of animal emoji between the brackets**. We added a blowfish, elephant, whale, camel, brontosaurus, kangaroo, and porcupine—but you can add whatever animals (or other emoji!) that you want.

Add a `;` after the closing curly bracket. This is what your statement should look like now:

A List is a collection that stores a set of values in order. You'll learn all about collections in Chapters 8 and 9.

When you use [brackets] to specify a List, you're using a collection expression. Each animal in quotes is a separate string value, and values are separated by commas. You'll learn more about expressions in Chapter 8.

```
List<string> animalEmoji = [
    "🐡", "🐡",
    "🐘", "🐘",
    "🐳", "🐳",
    "🐫", "🐫",
    "🐘", "🐘",
    "🐪", "🐪",
    "🦘", "🦘",
    "🦔", "🦔",
];
```

Be really careful with the quotes and commas. If you miss one, your code won't build.

Double-check that you're adding this code inside the { brackets } after the PlayAgainButton_Clicked method, and not the Button_Clicked method.

4 Finish the method.

You'll learn more about loops in the next chapter.

Add the rest of the code to add random animal emoji to the buttons—this code goes after the closing `];` at the end of the collection expression and before the `}` at the end of the method:

```
foreach (var button in AnimalButtons.Children.OfType<Button>())
{
    int index = Random.Shared.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    button.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

This is a *foreach loop*. It goes through a collection (like your list of emoji) and executes a set of statements for each item it finds.

Before you run your app, read through the code that you just added. It's OK if you don't understand everything that's going on with it yet. An important part of learning C# is starting to make the code make sense, and reading through it is a great way to do that.

Reading through C# code—even if you don't understand all of it yet—is a great way to make it all start to make sense.

5 **Make sure your code matches ours.**

Here's all of the C# code that you've added so far. We gave the parts that Visual Studio generated for you automatically a lighter color so you can see the code that you entered yourself.

```
namespace AnimalMatchingGame;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PlayAgainButton_Clicked(object sender, EventArgs e)
    {
        AnimalButtons.IsVisible = true;
        PlayAgainButton.IsVisible = false;

        List<string> animalEmoji = [
            "🐙", "🐙",
            "🐠", "🐠",
            "🐡", "🐡",
            "🦀", "🦀",
            "🦞", "🦞",
            "🦐", "🦐",
            "🦑", "🦑",
            "🐌", "🐌",
        ];

        foreach (var button in AnimalButtons.Children.OfType<Button>())
        {
            int index = Random.Shared.Next(animalEmoji.Count);
            string nextEmoji = animalEmoji[index];
            button.Text = nextEmoji;
            animalEmoji.RemoveAt(index);
        }
    }

    private void Button_Clicked(object sender, EventArgs e)
    {

    }
}
```

If you chose a different name for your project, this line will match that name.

You added this line to make the animal buttons invisible when the app first starts up.

Make sure there are exactly eight matching pairs of emoji. That's part of what makes the game work.

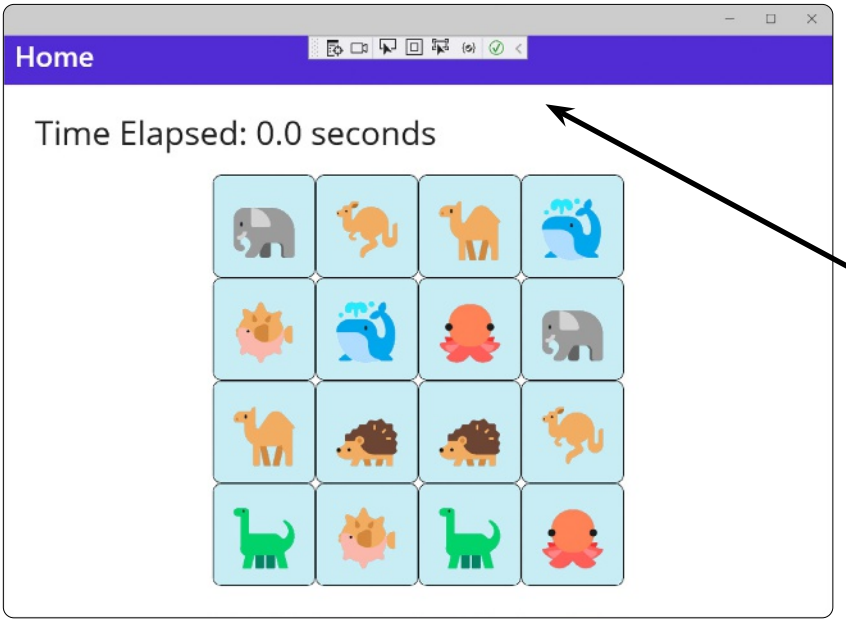
Visual Studio will *automatically indent* your code for you so it looks like ours. If you're using VSCode, press Alt-Shift-F or ⌥-Shift-F to automatically fix the indentation in your file.

You just added this code to add the emoji to the buttons.

Visual Studio added this empty `Button_Clicked` event handler method when you added a Clicked property to the button that you copied and pasted. *Make sure it's there!* If you're using VSCode, you might need to carefully type it by hand if it didn't get added automatically.

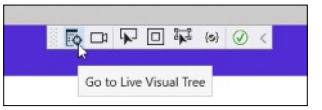
Run your app!

Run your app again. The first thing you'll see is the "Play again?" button. Click the button—you should now see eight pairs of animals in random positions:

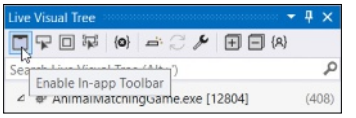


Stop it and run it again a few times. The animals should get reshuffled in a different order every time you click the "Play again?" button.

If you're using Visual Studio, you might see the *in-app toolbar* hovering at the top of the window:



We hid the in-app toolbar in our screenshots. You can keep yours visible or use the arrow at the right to collapse it.
You can also turn it off if you want (but you don't have to!). Click the first button in the runtime tools to bring up the Live Visual Tree panel in the IDE:



Then click the first button in the Live Visual Tree to enable or disable the in-app toolbar.

Wow, this game is already starting to look good!

You've set the stage for the next part that you'll add.

When you build a new game, you're not just writing code. You're also running a project. A really effective way to run a project is to build it in small increments, taking stock along the way to make sure things are going in a good direction. That way you have plenty of opportunities to change course.

This is a pencil-and-paper exercise. We included a lot of games and puzzles like this throughout the book. You should do all of them, because there's neuroscience evidence that writing things down is an effective way to get important concepts into your brain faster.

Who Does What?

C# statement	What it does
`List<string> animalEmoji = [` `];`	Make the button display the selected emoji Find every button in the FlexLayout and repeat the statements between the { curly brackets } for each of them Make the "Play again?" button invisible
`foreach (var button in AnimalButtons.Children.OfType<Button>())` `{`	Create a list of eight pairs of emoji
`animalEmoji.RemoveAt(index);`	Make the FlexLayout with the emoji buttons visible
`button.Text = nextEmoji;`	Pick a random number between 0 and the number of emoji left in the list and call it "index"
`string nextEmoji = animalEmoji[index];`	Remove the chosen emoji from the list
`AnimalButtons.IsVisible = true;`	Use the random number called "index" to get a random emoji from the list
`int index = Random.Shared.Next(animalEmoji.Count);`	
`PlayAgainButton.IsVisible = false;`	

Who Does What? Solution

C# statement

```
List<string> animalEmoji = [
    "🐶", "🐶",
    "🐸", "🐸",
    "🐘", "🐘",
    "🐰", "🐰",
    "🐫", "🐫",
    "🐦", "🐦",
    "🐦", "🐦",
    "🐢", "🐢",
    "🐢", "🐢",
];

foreach (var button in AnimalButtons.Children.OfType<Button>())
{

animalEmoji.RemoveAt(index);

button.Text = nextEmoji;

string nextEmoji = animalEmoji[index];

AnimalButtons.IsVisible = true;

int index = Random.Shared.Next(animalEmoji.Count);

PlayAgainButton.IsVisible = false;
```

What it does

Make the button display the selected emoji

Find every button in the FlexLayout and repeat the statements between the { curly brackets } for each of them

Make the "Play again?" button invisible

Create a list of eight pairs of emoji

Make the FlexLayout with the emoji buttons visible

Pick a random number between 0 and the number of emoji left in the list and call it "index"

Remove the chosen emoji from the list

Use the random number called "index" to get a random emoji from the list

Here's another pencil-and-paper exercise. Take a few minutes to do it!

MINI Sharpen your pencil

Here's a pencil-and-paper exercise that will help you **really start to understand** your C# code.

1. Take a piece of paper and turn it on its side so it's in landscape orientation, and draw a vertical line down the middle.
2. Write out the entire PlayAgainButton_Clicked method by hand on the left side of the paper, leaving space between each statement. (You don't need to be accurate with the emoji.)
3. On the right side of the paper, write each of the "what it does" answers above next to the statement that it's connected to. Read down both sides—it should all start to make sense.

I'm not sure about these "Sharpen your pencil" and matching exercises. Isn't it better to **just give me the code** to type into the IDE?

Working on your code comprehension skills will make you a better developer.

The pencil-and-paper exercises are **not optional**. They give your brain a different way to absorb the information. But they do something even more important: they give you opportunities to ***make mistakes***. Making mistakes is a part of learning, and we've all made plenty of mistakes (you may even find one or two typos in this book!). Nobody writes perfect code the first time—really good programmers always assume that the code that they write today will probably need to change tomorrow. In fact, later in the book you'll learn about *refactoring*, a name for programming techniques that are all about improving your code after you've written it.

We're serious—take the time to do the pencil-and-paper exercises. They're carefully designed to reinforce important concepts, and they're the fastest way to get the ideas in this book into your brain.

We'll add bullet points like this to give a quick summary of many of the ideas and tools that you've seen so far.

Bullet Points

- Visual Studio is Microsoft's **IDE**—or *integrated development environment*—that simplifies and assists in editing and managing your C# code files.
- .**Console apps** are cross-platform apps that use text for input and output.
- .NET **MAUI** (or .NET Multi-platform App UI) is a cross-platform framework for building visual apps in C#.
- MAUI user interfaces are designed in **XAML** (eXtensible Application Markup Language), an XML-based markup language that uses tags and properties to define controls in a user interface.
- MAUI apps are made up of **pages** that show **controls**.
- The FlexLayout control contains other controls and wraps them so they display on the page.
- The IDE's Properties window makes it easy to edit the properties of your controls like the text or font size.
- C# is made up of **statements** grouped into **methods**.
- An **event handler method** gets executed when specific events—like button clicks—happen.
- Visual Studio's AI-assisted **IntelliSense** and **IntelliCode** help you enter code more quickly.

You can use Git to save all of your code, and Visual Studio will help make it easy.

You're going to write a lot of code in this book! Wouldn't it be great if there was a convenient place to put that code so you can always go back to it?

We bet that you'll write some apps that you really like, and you'll want to share them with your friends so they can see the great things you've built.

Do you have a desktop and a laptop? A computer at home and at an office? Wouldn't it be great if you could start a project on one computer, then finish it on another one?

Imagine you're working on a project. You've spent hours getting the code right, and you're really happy with it. Then you make a few changes, and...oh no! Something went completely wrong, your code is broken, and you don't remember exactly what you changed. It would be great if you could see a history of all the changes you made, right?

Git can help you do all of those things!

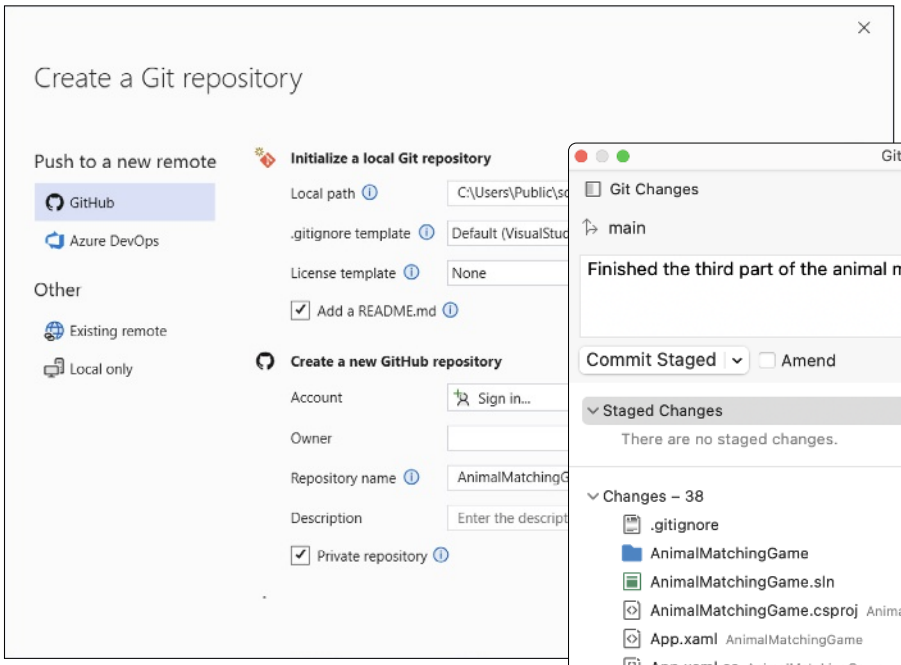
Here are just a few things Git can do for you

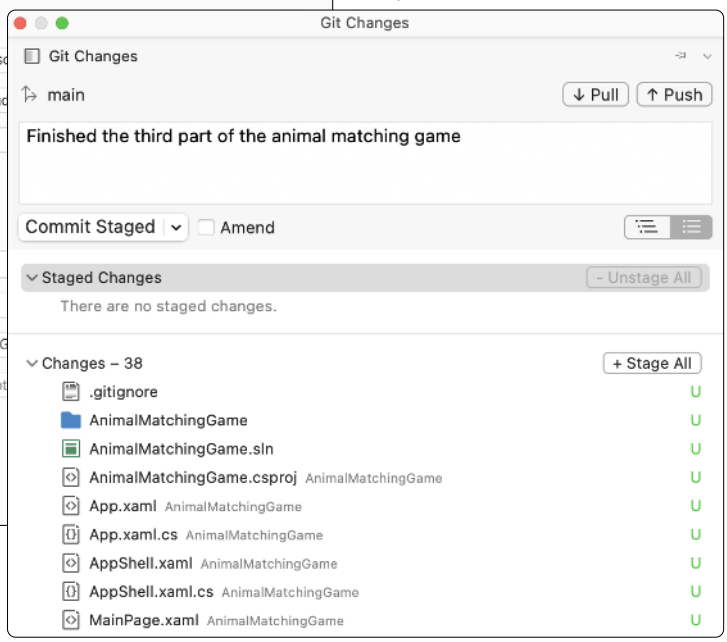
- It can save your files somewhere that you can access them from anywhere, any time.
- It lets you save snapshots of your work so you can go back and see exactly what changed.
- It lets you share your code with anyone (or keep it private!).
- It lets a group of people collaborate on a project together—so if you're learning C# with your friends, you can all work on code together.

Visual Studio makes it easy to use Git

Git is a really powerful and flexible tool that can help you save, manage, and share the code and files for all of your projects. It can also be complex and confusing at times! Luckily, Visual Studio has **built-in Git support** that takes care of the complexity. It helps you with Git, so you can concentrate on your code.

Visual Studio can help you create a new Git repository on GitHub, the popular platform for source code hosting and collaboration.



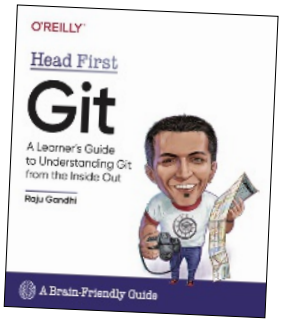


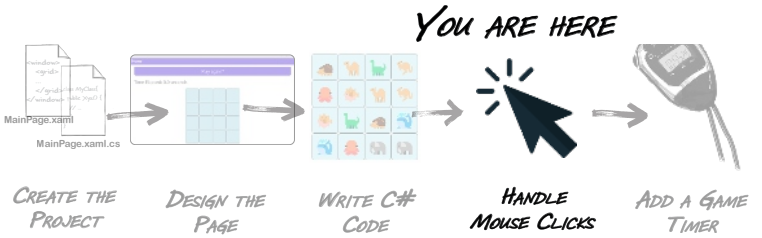
Visual Studio's Git features help you easily add your code to any Git and push changes as often as you want.

We recommend that you *create a GitHub account* and use it to save the code for each of the projects in this book. That will make it easy for you to go back and revisit past projects any time!

Our free Head First C# Guide to Git PDF gives you a simple, step-by-step guide to saving your code in Git with Visual Studio. Download it from *https://github.com/head-first-csharp/fifth-edition*.

We'll give you everything you need to use Visual Studio to save and share your projects. But there is a lot more that you can do with Git, especially if you're working with large teams! If you're fascinated by what you see and want to do a deep dive into Git, check out *Head First Git* by Raju Gandhi.



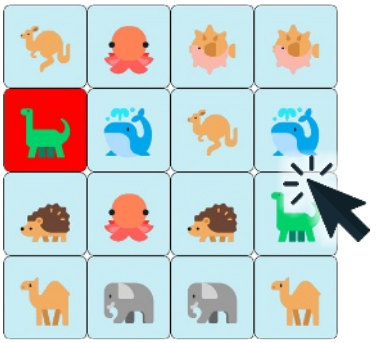


Add C# code to handle mouse clicks

You've got buttons with random animal emoji. Now you need them to do something when the player clicks them. Here's how it will work:

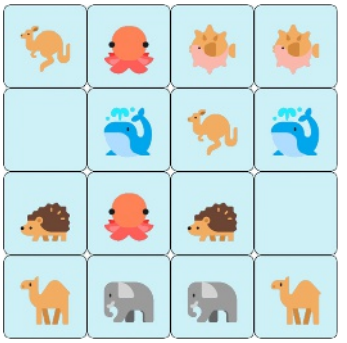
The player clicks the first button.

The player clicks buttons in pairs. When they click the first button, the game keeps track of that particular button's animal. The button that the player clicked changes color, so they can see what animal they clicked on.



The player clicks the second button.

The game looks at the animal on the second button and compares it against the first one they clicked on. The game compares its animal against the animal on the button that it kept track of from the first click.



The game checks for a match.

If the animals ***match***, the game goes through all of the emoji in its list of shuffled animal emoji. It finds any emoji in the list that match the animal pair the player found and replaces them with blanks.

If the animals ***don't match***, the game doesn't do anything.

In ***either case***, it resets its last animal found so it can do the whole thing over for the next click.

The game repeats this until all eight pairs of animals are matched.

Sharpen your pencil

When you added the Clicked event handler to your animal button, Visual Studio **automatically added a method called Button_Clicked** to *MainPage.xaml.cs*. Here's the code that will go into that method. Before you add this code to your app, read through it and try to figure out what it does.

We've asked you a few questions about what the code does. Try writing down the answers. ***It's OK if you're not 100% right!*** The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
Button lastClicked;
bool findingMatch = false;
int matchesFound;

private void Button_Clicked(object sender, EventArgs e)
{
  if (sender is Button buttonClicked)
  {
    if (!string.IsNullOrWhiteSpace(buttonClicked.Text) && (findingMatch == false))
    {
      buttonClicked.BackgroundColor = Colors.Red;
      lastClicked = buttonClicked;
      findingMatch = true;
    }
    else
    {
      if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
      {
        matchesFound++;
        lastClicked.Text = " ";
        buttonClicked.Text = " ";
      }
      lastClicked.BackgroundColor = Colors.LightBlue;
      buttonClicked.BackgroundColor = Colors.LightBlue;
      findingMatch = false;
    }
  }

  if (matchesFound == 8)
  {
    matchesFound = 0;
    AnimalButtons.IsVisible = false;
    PlayAgainButton.IsVisible = true;
  }
}
```

That's a double quote followed by a space and another double quote.

1. What does `matchesFound` do?

2. What do these three lines of code do?

3. What does this block of code do?

4. What do the last six lines of the method starting with `if (matchesFound == 8)` and going to the end do?

Sharpen your pencil Solution

We've asked you a few questions about what the code does. Try writing down the answers. ***It's OK if you're not 100% right!*** The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
Button lastClicked;
bool findingMatch = false;
int matchesFound;

private void Button_Clicked(object sender, EventArgs e)
{
  if (sender is Button buttonClicked)
  {
    if (!string.IsNullOrWhiteSpace(buttonClicked.Text) && (findingMatch == false))
    {
      buttonClicked.BackgroundColor = Colors.Red;
      lastClicked = buttonClicked;
      findingMatch = true;
    }
    else
    {
      if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
      {
        matchesFound++;
        lastClicked.Text = " ";
        buttonClicked.Text = " ";
      }
      lastClicked.BackgroundColor = Colors.LightBlue;
      buttonClicked.BackgroundColor = Colors.LightBlue;
      findingMatch = false;
    }
  }

  if (matchesFound == 8)
  {
    matchesFound = 0;
    AnimalButtons.IsVisible = false;
    PlayAgainButton.IsVisible = true;
  }
}
```

1. What does `matchesFound` do?

It keeps track of the number of pairs of animals the player found, so the game can end when they found all 8 pairs.

2. What do these three lines of code do?

These lines are run when the player clicks the first button of a potential match to change its color to red and keep track of it.

3. What does this block of code do?

This block of code is run when the player clicks on the second button in the pair. If the animals match, it adds one to matchesFound and blanks out the animals on both buttons. It also resets the color of the first button back and gets set for the player to click the first button in a pair again.

4. What do the last six lines of the method starting with `if (matchesFound == 8)` and going to the end do?

If matchesFound equals 8, the player found all 8 pairs of animals. When that happens, these lines reset the game by setting matchesFound back to zero, hiding the animal buttons, and showing the "Play again?" button so the player can start a new game by clicking the "Play again?" button.

Enter the code for the event handler

Did you do the "Sharpen your pencil" exercise? If not, take a few minutes and do it—you may not understand 100% of the code in the Button_Clicked event handler method yet, but you should at least have a basic sense of what's going on. And, more importantly, you've had a chance to look at it closely enough so that it should be familiar.

That familiarity will make it easier to **use the IDE to type the code into the method**. Stop your app if it's running—close the window or choose Stop Debugging (Shift+F5) from the Debug or Run menu—then edit *MainPage.xaml.cs*, find the Button_Clicked event handler method that Visual Studio added for you, and click on the line between its opening { and closing } curly brackets.

Now **start typing the code from the "Sharpen" solution** line by line. If you haven't used an IDE like Visual Studio or VSCode to write code, it may be a little weird seeing its IntelliSense and IntelliCode suggestions pop up. Use them if you can—the more you get used to them, the faster and easier it will be to write code later on in the book.

You need to be really careful when you're entering code, because if your opening parentheses or brackets don't have matches, or if you miss a semicolon at the end of a statement, your code won't build. Luckily, Visual Studio has a lot of features to help you write code that builds:

- When you enter `if` it automatically adds the opening and closing parentheses () so you don't accidentally leave them out.
- If you put your cursor in front of an opening parenthesis or bracket, it will highlight the closing one so you can easily see its match.
- A lot of the time, when you enter code that has problems—like writing `matchesFnd` instead of `matchesFound`, for example—it will often point out the error by drawing a red squiggly line underneath it.

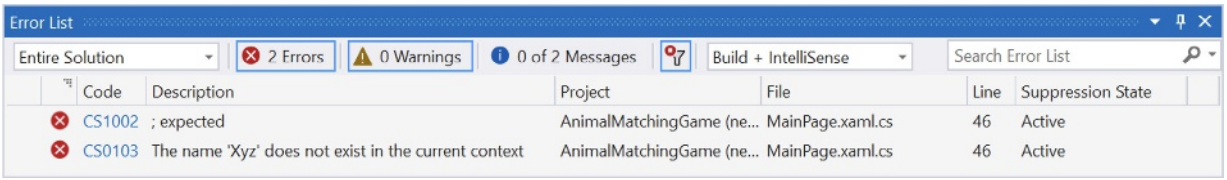
IDE Tip: The Error List

An operating system like Windows, macOS, Android, or iOS can't run C# code. That's why Visual Studio has to **build** your code, or turn it into a **binary** (a file that the operating system can run). Let's do an experiment and ***break your code***.

Go to the first line of code in your Button_Clicked method. Press Enter twice, then add this on its own line: `Xyz`

Check the bottom of the code editor again—you'll see an icon that looks like this: 2 or 2. If you don't see the icon, choose Build Solution from the Build menu to tell Visual Studio to try to build your code.

Click the icon (or choose Error List from the View menu) to open the Error List window. You'll see two errors in the window (if you're using a Mac it's called Errors and not Error List, and it looks a little different, but it displays the same information):



Visual Studio displayed these errors because `Xyz` is not valid C# code, and the errors prevent it from building your app. Your code won't run with those errors, so go ahead and delete the `Xyz` line that you added and build your app again.

If there are no other errors in your code, the Error List should be empty, and you'll see an icon that looks like this at the bottom of the Visual Studio window: No issues found or Build successful.—that tells you that your app builds.

Run your app and find all the pairs

Try running your app. If you entered all of the code correctly, it should start up and show you the "Play again?" button. Click the button to see a random list of animals. Then click each pair of animals one by one—each pair will disappear after you click it. Once you click the last pair of animals, the buttons will disappear and you'll see the "Play again?" button again.

Try experimenting with your app. Click mismatched pairs. Click in the window but outside the buttons. Click on the "Time elapsed" label. Click an empty button. Is your app working?

If your game doesn't work the way it should or you don't see the bug on this page, go back and check the code you entered against the code in the book. It's really easy to overlook a typo. Finding those issues is a good use of your time, because spotting errors in your code is a really good developer skill to work on.

Uh-oh—there's a bug in your code

If you typed in all of the code correctly, you may have noticed a problem. Start your app, click the "Play again?" button to show the random animals, and click on a pair to make the animals disappear from their buttons. Now **click the one of the blank buttons, then the other—and repeat that seven times**. Wait, what happened? Did the animal buttons disappear and the "Play again?" button appear, as if you'd won the game? That's not supposed to happen! Your game has a bug.

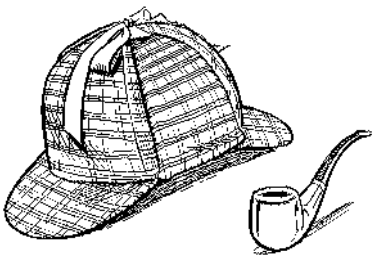
Don't worry, this bug is* not *your fault!

We left that bug in your code on purpose. You're going to be writing a lot of code throughout this book. Every chapter has several projects for you to work on…and there are opportunities for bugs in every one of those projects. Finding and fixing bugs is a normal and healthy part of writing code—and a really valuable skill for you to practice.

Every bug is caused by a problem in the code, so the first step in fixing a bug is figuring out what's causing it.

When you find a bug, you need to sleuth it out

Every bug is different. Code can break in many different ways. But there's one thing all bugs have common: every one of them ***is caused by a problem in the code***. So when there's a bug, your job is to figure out what's causing it, because you can't fix the problem until you know why it's happening.



If you've ever read a mystery novel or watched a detective show, you know that to solve a mystery, you need to **find the culprit**. So let's do that right now. It's time to put on your Sherlock Holmes cap, grab your magnifying glass, and **sleuth out what's causing the bug**.

Finding and fixing bugs is one part typing, nine parts thinking... and 100% guaranteed to make you a better developer. That's what these "Sleuth it Out" sections are all about.

Sleuth it Out

The Case of the Unexpected Match

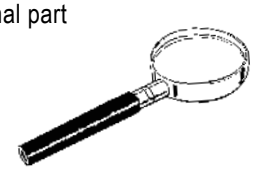
You've probably heard the word "bug" before.

You might have even said something like this to your friends at some point in the past: "That game is really buggy, it has so many glitches." Every bug has an explanation, and everything in your program happens for a reason…but not every bug is easy to track down. That's why we'll include tips for sleuthing out bugs throughout the book, starting with this "Sleuth it Out" section.

Every bug has a culprit.

Bugs are weird. They're what happens when your code does something you didn't expect it to do.

But bugs are also normal. Every developer spends time finding and fixing bugs. It's a normal part of writing code. You're going to write code that doesn't do what you expect it to. And when you do, the first thing you need to do is ***figure out what's causing the bug***.



The first step in finding a bug is thinking about what might have caused it.

Sherlock Holmes once said, "Crime is common. Logic is rare. Therefore it is upon the logic rather than upon the crime that you should dwell." That's great advice for figuring out what caused a bug. Don't get frustrated because your app doesn't do what you want (that's dwelling on the crime!). Instead, think about the logic of the situation. So let's look at the code and figure out what's going on.

Read the code carefully and search for clues.

We know that all of the code for handling mouse clicks is in the Button_Clicked event handler that you just added. So let's go back to the code and see if we can find clues about what went wrong.

Luckily, **you did that "Sharpen your pencil" exercise**. You looked closely at the code in the Button_Clicked event handler method to understand it. (If you haven't done that exercise yet, go back and do it now!)

Based on what we found in the "Sharpen your pencil" exercise, we already know a few things about the code:

- The event handler uses matchesFound to keep track of the number of pairs of animals the player found, so the game can end when they find all eight pairs.
- There's a part of the event handler that checks if the animals on the two buttons that the player clicked on match each other. If they do match, it adds one to matchesFound and blanks out both buttons.
- If matchesFound equals 8, the player found all 8 pairs of animals. There's code at the end of the event handler that checks to see if matchesFound is equal to 8, and if that's true it resets the game.

Those are the important clues that will help us find and fix the bug. Before you go on, can you sleuth out what's causing the game to end early if you keep clicking a button that's already been cleared?

Sleuth it Out

Why did the bug happen?

Let's think about those three clues for a minute. Here's what we know:

- The game uses matchesFound to keep track of the number of pairs of animals the player found.
- If the player clicks on a pair, the game increases matchesFound by 1 and blanks out the buttons the player clicked on.
- When matchesFound reaches 8, the game resets itself.

So what are these clues telling us? There's one conclusion that we can draw from these clues:

Somehow matchesFound is being increased by 1 when the player clicks on a button that's already blanked out.

Which means we have a starting point: the code that increases matchesFound by 1.

Go back to the scene of the crime

Here's the part of the code that increases matchesFound – the specific line that does that is in boldface:

```
if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
{
    matchesFound++;
    lastClicked.Text = " ";
    buttonClicked.Text = " ";
}
```

This statement uses the ++ operator to increase the value of matchesFound by 1. You'll learn about ++ and other operators in the next chapter.

The first line of code in the statements that we just showed you is an **`if` statement**, checks if something is true, and if it is then it executes statements. In this case, if the player clicked a different button than the first one in the pair (that's what "buttonClicked != lastClicked" checks for) and if the animals on those two buttons match ("buttonClicked.Text == lastClicked.Text"), it increases matchesFound by 1 and blanks out both buttons.

This is where things went wrong—which means it's also where we can fix the bug. We just need to find a way to keep matchesFound from getting increased by 1 **if the player clicked a button that's already blank**.

We found the culprit, so now we can fix the bug.

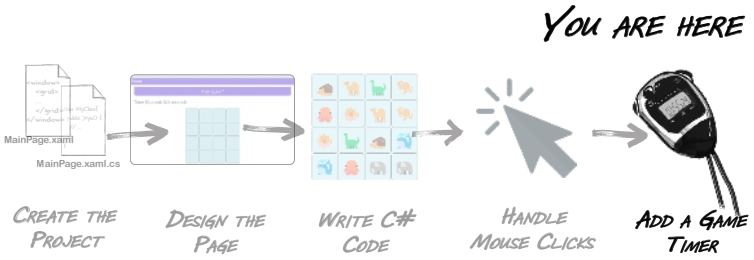
Position your cursor between the last two closing parentheses **`))`** in the **`if`** statement and press Enter to add a line. Then enter the following code: **`&& (!String.IsNullOrWhiteSpace(buttonClicked.Text))`**

Here's what your code should now look like:

```
if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text)
    && (!String.IsNullOrWhiteSpace(buttonClicked.Text)))
{
    matchesFound++;
    lastClicked.Text = " ";
    buttonClicked.Text = " ";
}
```

Adding this code to your "if" statement causes it to make sure the button that the player clicked on is not blank before adding 1 to matchesFound.

Once you've edited the **`if`** statement, run your app again. Now the bug should be fixed.



Finish the game by adding a timer

Our animal match game will be more exciting if players can try to beat their best time. We'll add a **timer** that "ticks" after a fixed interval by repeatedly calling a method.

Let's add some excitement to the game! The time elapsed since the game started will appear at the bottom of the window, constantly going up, and only stopping after the last animal is matched.

Timers "tick" every time interval by calling methods over and over again. You'll use a timer that starts when the player starts the game and ends when the last animal is matched.

Add a timer to your game's code

In this last part of your project, you'll add a timer to your game to make it more exciting. It will keep track of the time elapsed (in tenths of seconds), starting when the player clicks the "Play again?" button and stopping when they find the last match.

Do this!

1 Add a line of code to the end of the PlayAgainButton_Clicked event handler to start a timer.

Go to the very end of the PlayAgainButton_Clicked event handler. There are two closing curly brackets } at the end of the method on separate lines. Add three lines between the brackets, then add the following line of code into that space that you created:

```
    foreach (var button in AnimalButtons.Children.OfType<Button>())
    {
        int index = Random.Shared.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        button.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }

    Dispatcher.StartTimer(TimeSpan.FromSeconds(.1), TimerTick);
}
```

The line of code that you just added causes your app to **start a timer** that executes a method called TimerTick every 0.1 of a second.

2 Examine the error and click on "TimerTick" in the code you just added.

You just added a line of code to start a timer that "ticks" every 10th of a second. Every time it ticks, it calls a method called TimerTick. But hold on—your C# code doesn't have a TimerTick method. If you try to build your code, you'll see an error in the Error List window:

CS0103 The name 'TimerTick' does not exist in the current context

And there will be a red squiggly line underneath `TimerTick` in the line of code that you added. Click on `TimerTick` in the C# code—when you click on it, Visual Studio will display an icon shaped like a light bulb or screwdriver in the left margin.

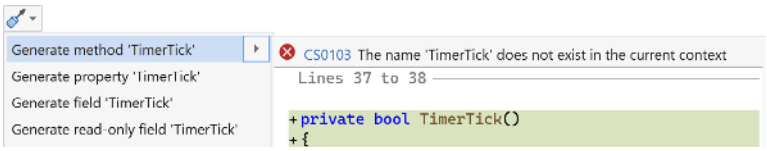
```
35            button.Text = nextEmoji;
36            animalEmoji.RemoveAt(index);
37        }
38
39        Dispatcher.StartTimer(TimeSpan.FromSeconds(.1), TimerTick);
40    }
41
```

When you click on TimerTick in the C# code, Visual Studio displays either a light bulb or screwdriver icon. It looks slightly different in Visual Studio Code, but works the same way.

The red squiggly line tells you under TimerTick that there's an error here.

③ Use Visual Studio to generate a new TimerTick method.

The code that you added has an error because it refers to a method called TimerTick that doesn't exist. When you click on it, a light bulb or screwdriver icon shows up in the lefthand margin. If you hover over it, you can see an error message and icon directly underneath it as well:



In VSCode, the Quick Actions icon looks like a light bulb, not a screwdriver.

Clicking the icon brings up the **Quick Actions menu**, which gives you some suggested potential fixes for the error. You can also click on TimerTick and press Alt+Enter or Ctrl+. on Windows or ⌘+. on a Mac—that's a Control or ⌘ plus period—to on a Mac to bring up the menu:

The first option in the Quick Actions menu should be "Generate method 'TimerTick'"—and if you select that option, you'll see a preview to the right. **Choose that option.**

Visual Studio will ***generate the TimerTick method for you***. Look through your C# code in *MainPage.xaml.cs* and find the TimerTick method that Visual Studio added:

```
private bool TimerTick()
{
    throw new NotImplementedException();
}
```

When your C# code has errors, Visual Studio sometimes has suggestions for potential fixes that can generate code to fix the error.

Finish the code for your game

In this last part of your project, you'll add a timer to your game to make it more exciting. It will keep track of the time elapsed (in tenths of seconds), starting when the player clicks the "Play again?" button and stopping when they find the last match.

Do this!

Add a field to hold the time elapsed

Find the first line of the TimerTick method that you just generated. Place your mouse cursor at the beginning of the line, then press Enter twice to add two spaces above it.

Add this line of code right above the TimerTick method you just added:

```
int tenthsOfSecondsElapsed = 0;

private bool TimerTick()
```

This is a field. You'll learn more about how fields work in Chapter 3.

Finish your TimerTick method

Now you have everything you need to finish the TimerTick method. Here's the code for it:

```
private bool TimerTick()
{
    if (!this.IsLoaded) return false;

    tenthsOfSecondsElapsed++;

    TimeElapsed.Text = "Time elapsed: " +
        (tenthsOfSecondsElapsed / 10F).ToString("0.0s");

    if (PlayAgainButton.IsVisible)
    {
        tenthsOfSecondsElapsed = 0;
        return false;
    }

    return true;
}
```

We put an extra line break in this statement so it would fit on the page in the printed book, but you can put it all on one line if you want. Make sure the parentheses match exactly.

Run your game. Now the timer works!

Your TimerTick Method Up Close

Let's take a closer look at your TimerTick method to see how it, well, ticks. It has a total of seven statements, and each of them is important.

```
private bool TimerTick()
{
    if (!this.IsLoaded) return false;

    tenthsOfSecondsElapsed++;

    TimeElapsed.Text = "Time elapsed: " +
        (tenthsOfSecondsElapsed / 10F).ToString("0.0s");

    if (PlayAgainButton.IsVisible)
    {
        tenthsOfSecondsElapsed = 0;
        return false;
    }

    return true;
}
```

If you close your app, the timer could still tick after the TimeElapsed label disappears, which could cause an error. This statement keeps that from happening.

The timer ticks every 10th of a second. Adding 1 to this field keeps track of how many of those 10ths have elapsed.

This statement updates the TimeElapsed label with the latest time, dividing the 10ths of second by 10 to convert it to seconds.

If the "Play Again?" button is visible again, that means the game is over and the timer can stop running. The `if` statement runs the next two statements only if the game is running.

We need to reset the 10ths of seconds counter so it starts at 0 the next time the game starts.

This statement causes the timer to stop, and no other statements in the method get executed.

This statement is only executed if the `if` statement didn't find the "Play again?" button visible. It tells the timer to keep running.

One last thing about the timer. The timer you used is guaranteed to fire no more than once every 10th of a second, but it may fire a little less frequently than that—which means the timer in the game may actually *run a little slow*. For this game, that's absolutely fine!

Even better ifs...

Your game is pretty good. Nice work! Every game—in fact, pretty much every program—can be improved. Here are a few things that we thought of that could make the game better:

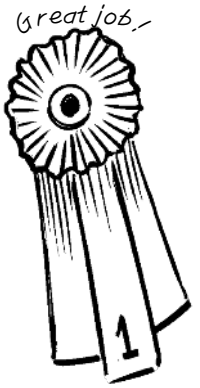
- Add different kinds of animals so the same ones don't show up each time.
- Keep track of the player's best time so they can try to beat it.
- Make the timer count down instead of counting up so the player has a limited amount of time.

MINI Sharpen your pencil

Can you think of your own "even better if" improvements for the game? This is a great exercise—take a few minutes and write down at least three improvements to the animal matching game.

We're serious—take a few minutes and do this. Stepping back and thinking about the project you just finished is a great way to seal the lessons you learned into your brain.

Congratulations—you built a game, but you did more than that! You took the time to really understand how it works, and that's a very important step in getting comfortable with C# concepts.



Did you add your code to a Git repo?

If you did, this is a great time to commit all of your changes and push it to the repository!

And if you still haven't, take a few minutes and check out our free ***Head First C# Guide to Git*** PDF. It gives you step-by-step instructions for keeping your code safe in Git.

Download it today from our own GitHub page: *https://github.com/head-first-csharp/fifth-edition*

Bullet Points

- An **event handler** is a method that your application calls when a specific event like a mouse click happens.
- Visual Studio makes it easy to **add and manage** your event handler methods.
- The IDE's **Error List window** shows any errors that prevent your code from building.
- A **timer** calls a method over and over again on a specified interval.
- **foreach** is a kind of loop that iterates through a collection of items.
- When you have a bug in your code, the first thing to do is try to **figure out what's causing it**.
- Bugs are normal, and sleuthing out bugs is an **important developer skill** that you'll work on throughout this book.
- Visual Studio makes it really easy to use **source control** to back up your code and keep track of all changes that you've made.
- You can commit your code to a remote **Git repository**. We use GitHub for the repository with the source code for all of the projects in this book.

2 variables, statements, and methods

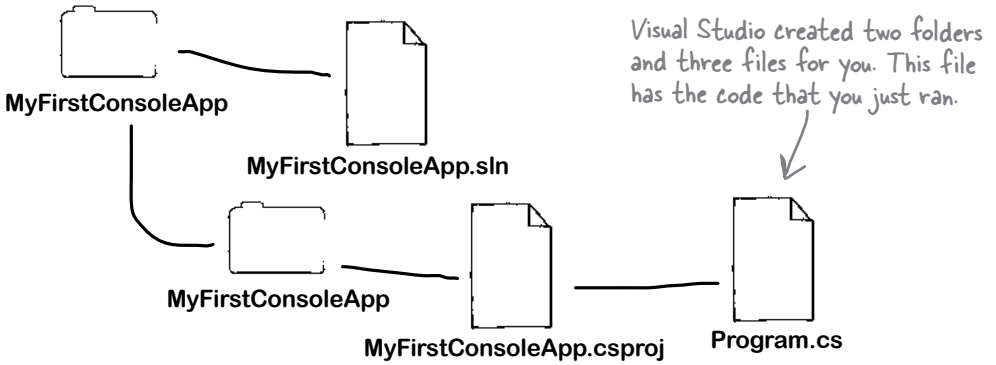
Dive into C# code

You're not just an IDE user. You're a developer.

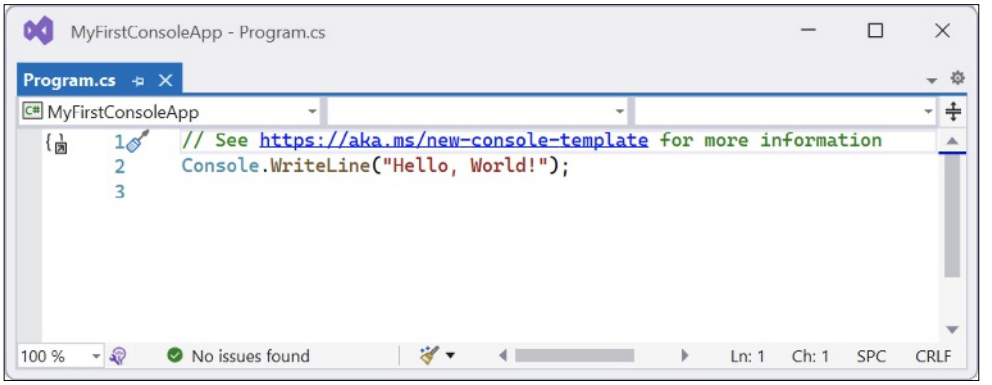
You can get a lot of work done using the IDE, but there's only so far it can take you. Visual Studio is one of the most advanced software development tools ever made, but a **powerful IDE** is only the beginning. It's time to ***dive* in to C# code**: how it's structured, how it works, and how you can take control of it…because there's no limit to what you can get your apps to do.

Take a closer look at the files in your console app

In Chapter 1, you created a new C# Console App project and named it MyFirstConsoleApp. When you did that, Visual Studio created two folders and three files.



Let's take a closer look at the *Program.cs* file that it created. Open it up in Visual Studio:



This is a screenshot of Visual Studio for Windows. If you're using VSCode the screen will look a little different, but the code will be the same.

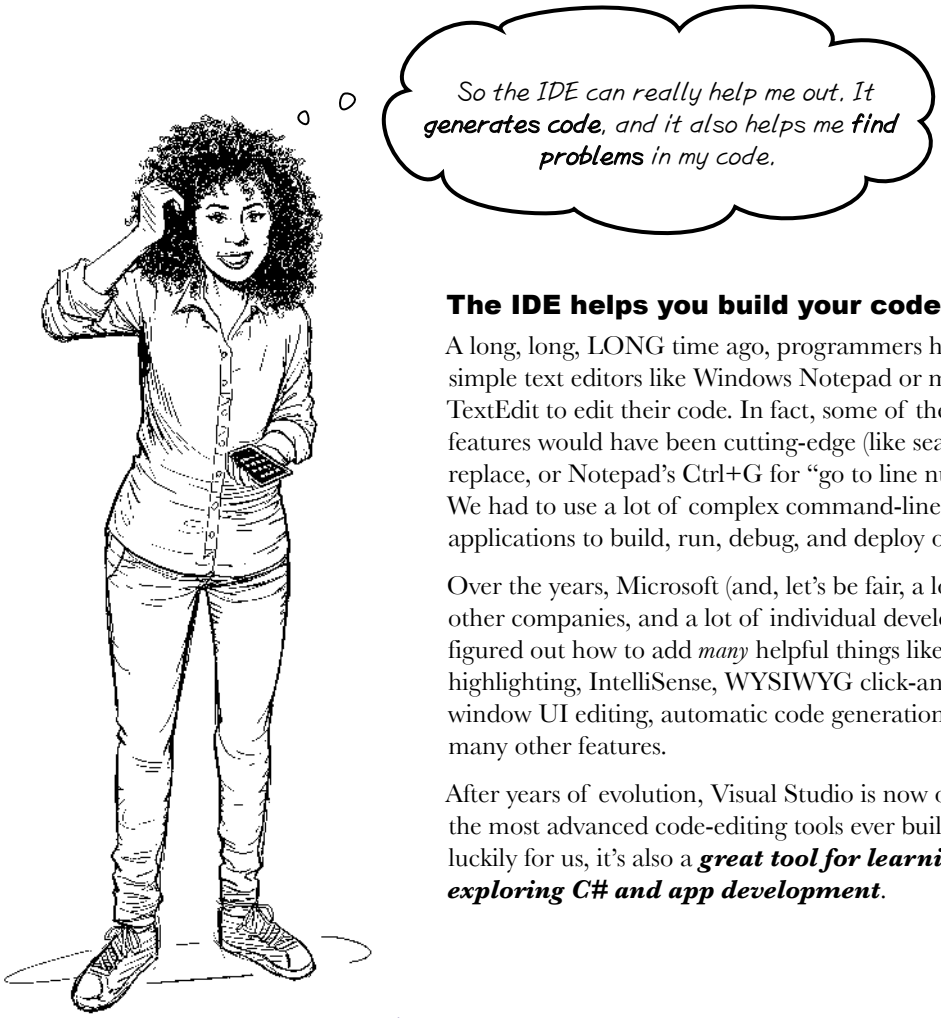
A statement performs one single action

A **console app** is an app with a text-only user interface. All its input and output goes to a console, like the Windows command prompt, the macOS Terminal, or Linux Terminal.

Your app has two lines:

1. The first line is a **comment**. Comments start with two forward slashes `//` and everything after those slashes is ignored. You can use comments to write notes about the code.
2. The second line is a **statement**. Statements are what make your code do things. In this case, it's a `Console.WriteLine` statement, which writes a line of text.

When you run your app, it starts with the first statement, and keeps executing statements until it runs out, and since it's a console app you'll see its output in a console window. Once it executes the last statements, the app exits.



The IDE helps you build your code right.

A long, long, LONG time ago, programmers had to use simple text editors like Windows Notepad or macOS TextEdit to edit their code. In fact, some of their features would have been cutting-edge (like search and replace, or Notepad's Ctrl+G for "go to line number"). We had to use a lot of complex command-line applications to build, run, debug, and deploy our code.

Over the years, Microsoft (and, let's be fair, a lot of other companies, and a lot of individual developers!) figured out how to add *many* helpful things like error highlighting, IntelliSense, WYSIWYG click-and-drag window UI editing, automatic code generation, and many other features.

After years of evolution, Visual Studio is now one of the most advanced code-editing tools ever built. And luckily for us, it's also a ***great tool for learning and exploring C# and app development***.

there are no Dumb Questions

Q: I understand what *Program.cs* does—that's where the code for my program lives. But does my program need the other two files and folders?

A: When you created a new project in Visual Studio, it created a **solution** for you. A solution is just a container for your project. The solution file ends in *.sln* and contains a list of the projects that are in the solution, with a small amount of additional information (like the version of Visual Studio used to create it). The **project** lives in a folder inside the solution folder. It gets a separate folder because some solutions can contain multiple projects—but yours only contains one, and it happens to have the same name as the solution (MyFirstConsoleApp). The project folder for your app contains two files: a file called *Program.cs* that contains the code, and a **project file** called *MyFirstConsoleApp.csproj* that has all of the information Visual Studio needs to **build** the code, which means turning it into something your computer can run. You'll eventually see **two more folders** underneath your project folder: the **bin folder** will have the executable files built from your C# code, and the **obj folder** will have the temporary files used to build it.

Statements are the building blocks for your apps

Your app is made up of classes, and those classes contain methods, and those methods contain statements. A **statement** is a line of code that does something.

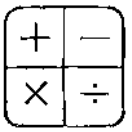
A statement can actually span multiple lines, which you'll see later in this book. But for now, you can just think of "statement" and "line of code" as the same thing.

So if we want to build apps that do a lot of things, we'll need a few **different kinds of statements** to make them work. You've already seen one kind of statement:

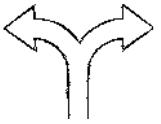
```
Console.WriteLine("Hello, World!");
```

This is a **statement that calls a *method***—specifically, the Console.WriteLine method, which prints a line of text to the console. We'll also use a few other kinds of statements in this chapter and throughout the book. For example:

We use variables and variable declarations to let our app store and work with data.



Lots of programs use math, so we use mathematical operators to add, subtract, multiply, divide, and more.



Conditionals let our code choose between options, either executing one block of code or another.

Loops let our code run the same block over and over again until a condition is satisfied.

Statements live inside methods

You wrote a **method** in Chapter 1 to set up your animal matching game. But what, exactly, is a method?

Methods do something

The Console.WriteLine method is part of .NET. It's not hard to guess that a method that starts with "Console." has something to do with reading or writing text in a console app. In this case, it writes a line of text to the console. It's a really useful method, and you'll use it—and a lot of other .NET methods (it has *thousands* of them!)—throughout this book.

You're going to write your own methods, and you're going to write code that **calls** those methods. To call a method, you write a statement that consists of the name of that method followed by parentheses and a semicolon. You can **pass** information to that method by putting it inside those parentheses—like passing "Hello, World!" when your code called the Console.WriteLine method.

Methods help you organize your code

Every method is made up of statements, and one method can contain many statements. Code tends to naturally organize into **blocks**, or lines of code which, taken together, do a specific thing. Methods are your way to take those code blocks, give them names, and make them easy to call.

When your program calls a method, it executes the first statement in that method, then the next, then the next, etc. When the method runs out of statements—or hits a **`return`** statement—it ends, and the program execution resumes after the statement that originally called the method.

Do you really *need* a method? You *could* copy the code in a method and paste it over the statement that called that method, and the app would still work. When you put a block of code into a method and ***give it a name***, you make it a lot easier to understand what that code does.

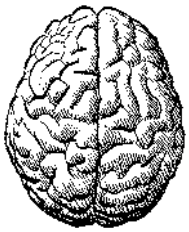
Brain Power

You'll use methods over and over again throughout this book to organize your code. Why do you think your code needs organizing?

When you're writing your code, you can take a block of code and turn it into a single method, multiple methods, or not use methods at all. How do you decide where to break up your code into methods?

The Console.WriteLine method writes a line to the console. Does that name make sense to you? Can you think of why it's useful for methods to have sensible names?

Your methods use variables to work with data



Every program, no matter how big or how small, works with data. Sometimes the data is in the form of a document, or an image in a video game, or a social media update—but it's all just data. That's where **variables** come in. A variable is what your program uses to store data.

Declare your variables

Whenever you **declare** a variable, you tell your program its *type* and its *name*. Once C# knows your variable's type, it will generate errors that stop your program from building if you try to do something that doesn't make sense, like subtract `"Fido"` from `48353`. Here's how to declare variables:

```
// Let's declare some variables
int maxWeight;
string message;
bool boxChecked;
```

Any line that starts with // is a *comment* and does not get executed. You can use comments to add notes to your code to help people read and understand it.

These are variable types. C# uses the *type* to define what data these variables can hold.

These are variable *names*. C# doesn't care what you name your variables—these names are for you.

This is why it's really helpful for you to choose variable names that make sense and are obvious.

Variables vary

A variable is equal to different values at different times while your program runs. In other words, a variable's value ***varies***. (Which is why "variable" is such a good name.) This is really important because that idea is at the core of every program you'll write. Say your program sets the variable `myHeight` equal to 63:

```
int myHeight = 63;
```

Any time `myHeight` appears in the code, C# will replace it with its value, 63. Then, later on, if you change its value to 12:

```
myHeight = 12;
```

C# will replace `myHeight` with 12 from that point onward (until it gets set again)—but the variable is still called `myHeight`.

Whenever your program needs to work with numbers, text, true/false values, or any other kind of data, you'll use variables to keep track of them. The variable's type defines what kind of data it can hold.

You need to assign values to variables before you use them

Try typing these statements just below the "Hello, World" statement in your new console app:

Do this!

```
string z;
string message = "The answer is " + z;
```

Go ahead, try it right now. You'll get an error, and the IDE will refuse to build your code. That's because it checks each variable to make sure that you've assigned it a value before you use it. The easiest way to make sure you don't forget to assign values to your variables is to combine the statement that declares a variable with a statement that assigns its value:

```
int maxWeight = 25000;
string message = "Hi!";
bool boxChecked = true;
```

These values are assigned to the variables. You can declare a variable and assign its initial value in a single statement (but you don't have to).

If you write code that uses a variable that hasn't been assigned a value, your code won't build. It's easy to avoid that error by combining your variable declaration and assignment into a single statement.

Once you've assigned a value to your variable, that value can change. So there's no disadvantage to assigning a variable an initial value when you declare it.

A few useful types

Every variable has a type that tells C# what kind of data it can hold. We'll go into a lot of detail about the many different C# types in Chapter 4. In the meantime, we'll concentrate on the three most popular types. `int` holds integers (or whole numbers), `string` holds text, and `bool` holds **Boolean** true/false values.

var-i-a-ble, noun.
an element or feature likely to change.
Predicting the weather would be a whole lot easier if meteorologists didn't have to take so many ***variables*** *into account.*

Generate a new method to work with variables

In Chapter 1, you learned that Visual Studio will **generate code for you**. This is quite useful when you're writing code—and ***it's also a really valuable learning tool***. Let's build on what you learned and take a closer look at generating methods.

Do this!

1 Add a method to your new MyFirstConsoleApp project.

Open the Console App project that you created in the previous chapter. In **Visual Studio** choose File >> Open >> Project/Solution. In **VSCode**, choose File >> Close Folder, then File >> Open Folder.

The *Program.cs* file has two lines of code that we saw in Chapter 1. **Replace those two lines** with this statement—it ***calls a method named*** `OperatorExamples`:

```
OperatorExamples();
```

A statement with the name of a method followed by opening and closing parentheses () calls that method.

2 Let Visual Studio tell you what's wrong.

As soon as you finish replacing the statements, Visual Studio will draw a red squiggly underline beneath your method call. Hover your mouse cursor over it. The IDE will display a pop-up window:

```
OperatorExamples();
```

CS0103: The name 'OperatorExamples' does not exist in the current context

Show potential fixes (Alt+Enter or Ctrl+.)

In VSCode, the box may look a little different, and instead of "potential fixes" it may say Quick Fix instead.

Visual Studio is telling you two things: that there's a problem—you're trying to call a method that doesn't exist (which will prevent your code from building)—and that it has a **potential fix**.

3 Generate the OperatorExamples method.

In **Visual Studio**, the pop-up window tells you to press Alt+Enter or Ctrl+. to see the potential fixes. In **VSCode**, you may see a "Quick fix" link—press Ctrl+. / ⌘+. to see the potential fixes. Click on OperatorExamples and click the icon that looks like a screwdriver or light bulb (or press Ctrl+. or ⌘+.).

```
OperatorExamples();
```

Generate method 'OperatorExamples'

Introduce local for 'OperatorExamples()'

★ IntelliCode suggestion based on recent edits: OperatorExamples()

CS0103 The name 'OperatorExamples' does not exist in the current context

```
OperatorExamples();

void OperatorExamples()
{
    throw new NotImplementedException();
}
```

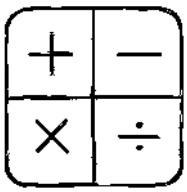
Preview changes

When the IDE generates a new method for you, it adds this "throw" statement as a placeholder—if you run your program, it will halt as soon as it hits that statement. You'll replace that "throw" statement with code.

This screenshot is from Windows. VSCode may not show you a preview of the generated code.

Choose the option to generate the method. The IDE will add three lines of code to *Program.cs* that match the code in the preview window. Your top-level statements now have an OperatorExample method.

Add code that uses operators to your method



Once you've got some data stored in a variable, what can you do with it? Well, if it's a number, you might want to add or multiply it. If it's a string, you might join it together with other strings. That's where operators come in. Here's the **method body** for your new OperatorExamples method. **Add this code to your program**, and read the comments to learn about the operators it uses.

```
void OperatorExamples()
{
  // This statement declares a variable and sets it to 3
  int width = 3;

  // The ++ operator increments a variable (adds 1 to it)
  width++;

  // Declare two more int variables to hold numbers and
  // use the + and * operators to add and multiply values
  int height = 2 + 4;
  int area = width * height;
  Console.WriteLine(area);

  // The next two statements declare string variables
  // and use + to concatenate them (join them together)
  string result = "The area";
  result = result + " is " + area;
  Console.WriteLine(result);

  // A Boolean variable is either true or false
  bool truthValue = true;
  Console.WriteLine(truthValue);
}
```

When you generated the OperatorExamples method, the IDE generated code that included the statement `throw new NotImplementedException();` You should *delete* that statement and *replace* it with the code we've given you, so your OperatorExample method *matches our code exactly*. Be very careful that all of the quotes, parentheses, math symbols, and semicolons match—it's easy to make a typo.

String variables hold text. When you use the + operator with strings, it joins them together. Adding "abc" + "def" results in a single string, "abcdef."

Sharpen your pencil

The statements you just added to your code will write three lines to the console: each `Console.WriteLine` statement prints a separate line. **Before you run your code**, figure out what they'll be and write them down. And don't bother looking for a solution, because we didn't include one! Just run the code to check your answers.

Here's a hint: converting a bool to a string results in either False or True.

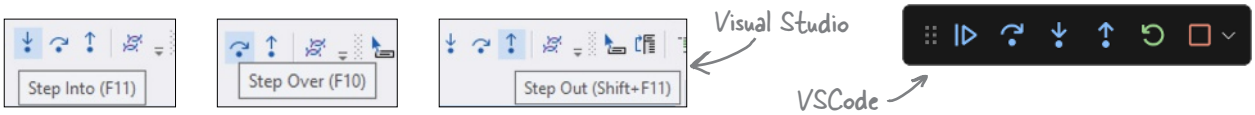
Line 1: _____

Line 2: _____

Line 3: _____

Use the debugger to watch your variables change

When you ran your program earlier, it was executing in the **debugger**—and that's an incredibly useful tool for understanding how your programs work. You can use **breakpoints** to pause your program when it hits certain statements and add **watches** to look at the value of your variables. Let's use the debugger to see your code in action. We'll use these three features of the debugger, which you'll find in the toolbar:



If you end up in a state you don't expect, just use the Restart button ( ) to restart the debugger.

Debug this!

1 **Add a breakpoint and run your program.**

Click on the first line of your program and **press F9 to add a breakpoint**. The line should now look like this—the line should be highlighted in red with a dot in the left margin:

```
1  OperatorExamples();
2
3  void OperatorExamples()
```

The screens in VSCode will look slightly different, but the debugger works the same way, with the same shortcut keys.

Then press the MyFirstConsoleApp button to run your program in the debugger, just like you did earlier.

If your app doesn't pause on the breakpoint, make sure you're starting the app with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu.

2 **Step into the method.**

Your debugger is stopped at the breakpoint on the statement that calls the OperatorExamples method.

```
1  OperatorExamples();
2
3  void OperatorExamples()
4  {
```

The red background and dot show you where you've set breakpoints. The yellow arrow and highlight show the line of code the debugger is paused on.

Press Step Into (F11)—the debugger will jump into the method and pause before it runs the first statement.

3 **Examine the value of the width variable.**

When you're **stepping through your code**, the debugger pauses after each statement that it executes. This gives you the opportunity to examine the values of your variables. Hover over the `width` variable.

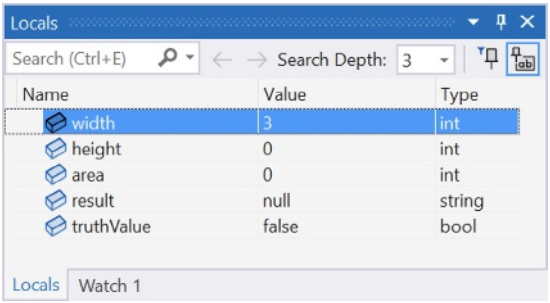
```
3  void OperatorExamples()
4  {   2ms elapsed
5      // This statement declares a variable
6      int width = 3;
7              width   0
```

The highlighted bracket and arrow in the left margin mean the code is paused just before the first statement of the method.

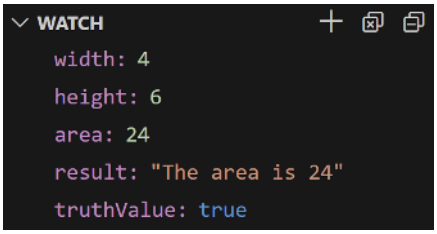
The IDE displays a pop-up that shows the current value of the variable—it's currently 0. Now **press Step Over (F10)**—it goes past the comment to the first statement, which is now highlighted. Now **press Step Over again**, then hover over `width` again. It now has a value of 3.

4 **The Locals window shows the values of your variables.**

The variables that you declared are **local** to your OperatorExamples method—which just means that they exist only inside that method, and can only be used by statements in the method. Visual Studio displays their values in the Locals window at the bottom of the IDE when it's debugging.



Visual Studio

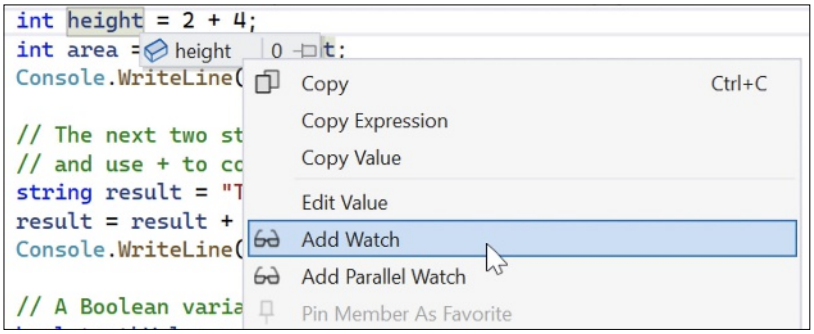


VSCode

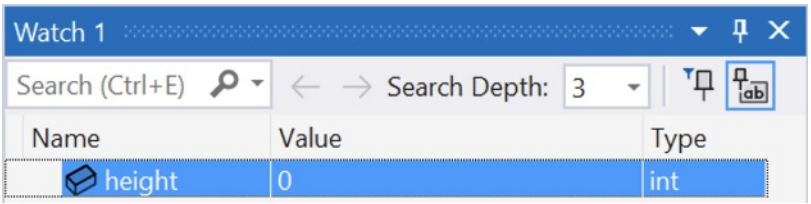
VSCode doesn't have a Locals window yet, but the Watch window works just like the one in Visual Studio.

5 **Add a watch for the height variable.**

A really useful feature of the debugger is the **Watch window**, which is typically in the same panel as the Locals window at the bottom of the IDE. When you hover over a variable, you can add a watch by right-clicking on the variable name in the pop-up window and choosing Add Watch. Hover over the `height` variable, then right-click and choose **Add Watch** from the menu.



Now you can see the `height` variable in the Watch window.



The debugger is one of the most important features in Visual Studio, and it's a great tool for understanding how your programs work.

6 **Step through the rest of the method.**

Step over each statement in OperatorExamples. As you step through the method, keep an eye on the Locals or Watch window and watch the values as they change. On **Windows**, press **Alt+Tab** before and after the `Console.WriteLine` statements to switch back and forth to the Debug Console to see the output. On **macOS**, you'll see the output in the Terminal window so you don't need to switch windows.

Use code snippets to help write loops

At the time we're writing this, VSCode does not support C# code snippets.

Do this!

You'll be writing a lot of loops throughout this book, and Visual Studio can help speed things up for you with **snippets**, or simple templates that you can use to add code. Let's use snippets to add a few loops to your OperatorExamples method.

If your code is still running, choose **Stop Debugging** (**Shift+F5**) from the Debug menu (or press the Stop button ■ in the toolbar). Then find the line `Console.WriteLine(area);` in your OperatorExamples method. Click at the end of that line so your cursor is after the semicolon, then press Enter a few times for some space. Now start your snippet. Press Ctrl+K, Ctrl+X (so hold down control and press K and then X) to bring up the Insert Snippet box. Choose "Visual C#" and then **type or click `while` and press Enter.** The IDE will add a template for a `while` loop to your code, with the conditional test highlighted:

```
while (true)
{

}
```

IDE Tip: Brackets

If your brackets (or braces, either name will do) don't match up, your program won't build, which leads to frustrating bugs. Luckily, the IDE can help with this! Put your cursor on a bracket, and the IDE highlights its match.

Type **`area < 50`**—the IDE will replace `true` with the text. **Press Enter** to finish the snippet. Then add two statements between the brackets:

```
while (area < 50)
{
    height++;
    area = width * height;
}
```

You can also type in this code instead of using a snippet. Visual Studio Code may not support snippets.

Next, use the **`do/while` loop snippet** to add another loop immediately after the while loop you just added. Press Ctrl+K, Ctrl+X and choose the C# do snippet. The IDE will add this snippet:

```
do
{

} while (true);
```

Again, you can just type in the code instead of using a snippet.

For most snippets you can type the snippet name and press tab twice. Go to a blank line and try it—type the word "do" and press the tab key twice.

Type `area > 25` and press Enter to finish the snippet. Then add two statements between the brackets:

```
do
{
    width--;
    area = width * height;
} while (area > 25);
```

Now **use the debugger** to really get a good sense of how these loops work:

1. Click on the line just above the first loop and choose **Toggle Breakpoint** (**F9**) from the Debug menu to add a breakpoint. Then run your code and **press F5** to skip to the new breakpoint.
2. Use **Step Over** (**F10**) to step through the two loops. Watch the Locals window as the values for `height`, `width`, and `area` change.
3. Stop the program, then change the `while` loop test to **`area < 20`** so both loops have conditions that are false. Debug the program again. The `while` checks the condition first and skips the loop, but the `do/while` executes it once and then checks the condition.

Use operators to work with variables

Once you have data in a variable, what do you do with it? Well, most of the time you'll want your code to do something based on the value. That's where **equality operators**, **relational operators**, and **logical operators** become important.

Equality Operators

The == operator compares two things and is true if they're equal.

The != operator works a lot like ==, except it's true if the two things you're comparing are not equal.

Relational Operators

Use > and < to compare numbers and see if a number in one variable is bigger or smaller than another.

You can also use >= to check if one value is greater than or equal to another, and <= to check if it's less than or equal.

Logical Operators

You can combine individual conditional tests into one long test using the && operator for ***and*** and the || operator for ***or***.

Here's how you'd check if **`i`** equals 3 ***or*** **`j`** is less than 5:

```
(i == 3) || (j < 5)
```

Watch it!

Don't confuse the two equals sign operators!

You use one equals sign (=) to set a variable's value, but two equals signs (==) to compare two variables. You won't believe how many bugs in programs—even ones made by experienced programmers!—are caused by using = instead of ==. If you see the IDE complain that you "cannot implicitly convert type 'int' to 'bool,'" that's probably what happened.

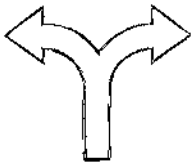
Use operators to compare two int variables

You can do simple tests by checking the value of a variable using a comparison operator. Here's how you compare two ints, x and y:

```
x < y (less than)
x > y (greater than)
x == y (equals - and yes, with two equals signs)
```

These are the ones you'll use most often.

if statements make decisions



Use **`if` statements** to tell your program to do certain things only when the **conditions** you set up are (or aren't) true. The `if` statement ***tests the condition*** and executes code if the test passes. A lot of `if` statements check if two things are equal. That's when you use the `==` operator. That's different from the single equals sign (=) operator, which you use to set a value.

```
int someValue = 10;
string message = "";

if (someValue == 24)
{
    message = "Yes, it's 24!";
}
```

Every `if` statement starts with a test in parentheses, followed by a *block* of statements in brackets to execute if the test passes.

The statements inside the curly brackets are executed only if the test is true.

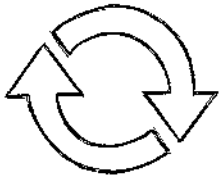
if/else statements also do something if a condition isn't true

`if/else` statements are just what they sound like: if a condition is true they do one thing ***or else*** they do the other. An `if/else` statement is an `if` statement followed by the **`else` keyword** followed by a second set of statements to execute. If the test is true, the program executes the statements between the first set of brackets. Otherwise, it executes the statements between the second set.

```
if (someValue == 24)
{
  // You can have as many statements
  // as you want inside the brackets
  message = "The value was 24.";
}
else
{
  message = "The value wasn't 24.";
}
```

REMEMBER—always use two equals signs to check if two things are equal to each other.

Loops perform an action over and over



Here's a peculiar thing about most programs (*especially* games!): they almost always involve doing certain things over and over again. That's what **loops** are for—they tell your program to keep executing a certain set of statements as long as some condition is true or false.

while loops keep looping statements while a condition is true

In a `while` **loop**, all of the statements inside the curly brackets get executed as long as the condition in the parentheses is true:

```
while (x > 5)
{
  // Statements between these brackets will
  // only run if  x is greater than 5, then
  // will keep looping as long as x > 5
}
```

do/while loops run the statements then check the condition

A `do/while` loop is just like a `while` loop, with one difference. The `while` loop does its test first, then runs its statements only if that test is true. The `do/while` loop runs the statements first, ***then*** runs the test. So if you need to make sure your loop always runs at least once, a `do/while` loop is a good choice:

```
do
{
  // Statements between these brackets will run
  // once, then keep looping as long as x > 5
} while (x > 5);
```

for loops run a statement after each loop

A `for` **loop** runs a statement after each time it executes a loop.

Every `for` loop has *three statements*. The first statement sets up the loop. It will keep looping as long as the second statement is true. The third statement gets executed after each time through the loop.

```
for (int i = 0; i < 8; i = i + 2)
{
    // Everything between these brackets
    // is executed four times
}
```

The parts of the `for` statement are called the *initializer* (int i = 0), the *conditional test* (i < 8), and the *iterator* (i = i + 2). Each time through a `for` loop (or any loop) is called an *iteration*.
The conditional test always runs at the beginning of each iteration, and the iterator always runs at the end of the iteration.

for Loops Up Close

A **for loop** is a little more complex than a **while** loop or **do** loop. The most common type of **for** loop just counts up to a length. The **for code snippet** causes the IDE to create an example of that kind of **for** loop:

```
for (int i = 0; i < length; i++)
{

}
```

When you use the "for" snippet, press Tab to switch between i and length. If you change the name of the variable i, the snippet will automatically change the other two occurrences of it.

A **for** loop has four sections—an initializer, a condition, an iterator, and a body:

```
for (initializer; condition; iterator) {
    body
}
```

Most of the time you'll use the initializer to declare a new variable—for example, the initializer **int i = 0** in the previous **for** code snippet declares a variable called **i** that can only be used inside the **for** loop. The loop will then execute the body—which can either be one statement or a block of statements inside curly braces—as long as the condition is true. At the end of each iteration, the for loop executes the iterator. So this loop:

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine("Iteration #" + i);
}
```

will iterate 10 times, printing **Iteration #0**, **Iteration #1**, ..., **Iteration #9** to the console.

Sharpen your pencil

Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop? Also, answer the questions in the comments in loops #2 and #3.

```
// Loop #1
int count = 5;
while (count > 0) {
   count = count * 3;
   count = count * -1;
}
```

Remember, a for loop always runs the conditional test at the beginning of the block, and the iterator at the end of the block.

```
// Loop #4
int i = 0;
int count = 2;
while (i == 0) {
   count = count * 3;
   count = count * -1;
}
```

```
// Loop #2
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
       // How many times will
       // the next statement
       // be executed?
       j = j + 5;
    }
}
```

```
// Loop #5
while (true) { int i = 1;}
```

```
// Loop #3
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      // How many times will
      // the next statement
      // be executed?
      p = p * 2;
   }
   q = p - q;
}
```

Hint: p starts out equal to 2. Think about when the statement "p = p * 2" is executed.

Some useful things to keep in mind about C# code

- **Don't forget that all your statements need to end in a semicolon.**
  `name = "Joe";`

- **Add comments to your code by starting a line with two slashes.**
  `// This text is ignored`

- **Use /* and */ to start and end comments that can include line breaks.**
  ```
  /* This comment
   * spans multiple lines */
  ```

- **Variables are declared with a *type* followed by a *name*.**
  `int weight;`
  `// The variable's type is int and its name is weight`

- **Most of the time, extra whitespace is fine.**
  So this: `int j = 1234 ;`
  is exactly the same as this: `int j = 1234;`

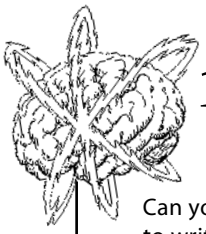
- **If/else, while, do, and for are all about testing conditions.**
  Every loop we've seen so far keeps running as long as a condition is true.

*There's a **flaw in your logic**! What happens if I write a loop with a conditional test that **never becomes false**?*

Then your loop runs forever.

Every time your program runs a conditional test, the result is either **`true`** or **`false`**. If it's **`true`**, then your program goes through the loop one more time. Every loop should have code that, if it's run enough times, should cause the conditional test to eventually return **`false`**. If it doesn't, then the loop will keep running until you kill the program or turn the computer off!

This is sometimes called an infinite loop, and there are definitely times when you'll want to use one in your code.



Brain Power

Can you think of a reason that you'd want to write a loop that never stops running?

When we give you pencil-and-paper exercises, we'll usually give you the solution on the next page.

Sharpen your pencil Solution

Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop? Also, answer the questions in the comments in loops #2 and #3.

```
// Loop #1
int count = 5;
while (count > 0) {
   count = count * 3;
   count = count * -1;
}
```

Loop #1 executes once.

Remember, count = count * 3 multiplies count by 3, then stores the result (15) back in the same count variable.

```
// Loop #4
int i = 0;
int count = 2;
while (i == 0) {
   count = count * 3;
   count = count * -1;
}
```

Loop #4 runs forever.

```
// Loop #2
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
       // How many times will
       // the next statement
       // be executed?
       j = j + 5;
    }
}
```

Loop #2 executes seven times.

The statement j = j + 5 is executed 6 times.

```
// Loop #5
while (true) { int i = 1;}
```

Loop #5 is also an infinite loop.

```
// Loop #3
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      // How many times will
      // the next statement
      // be executed?
      p = p * 2;
   }
   q = p - q;
}
```

Loop #3 executes eight times.

The statement p = p * 2 executes three times.

Remember, if your app doesn't pause on the breakpoint, make sure you're starting the app with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu.

Take the time to *really figure out* how loop #3 works. Here's a perfect opportunity to try out the debugger on your own! Set a breakpoint on `q = p - q;` and *add watches for p and q* to keep track of how the values of `p` and `q` change as you step through the loop.

Sharpen your pencil

Let's get some practice working with conditionals and loops. Update the code in your Program.file to match the following new code, including TryAnIf, TryAnIfElse, and TrySomeLoops methods. ***Before you run your code***, read it carefully and try to answer the questions based on how you think it will run. Then run your code and see if you got them right.

```
TryAnIf();
TrySomeLoops();
TryAnIfElse();

static void TryAnIf()
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        Console.WriteLine("x is 3 and the name is Joe");
    }
    Console.WriteLine("this line runs no matter what");
}

static void TryAnIfElse()
{
    int x = 5;
    if (x == 10)
    {
        Console.WriteLine("x must be 10");
    }
    else
    {
        Console.WriteLine("x isn't 10");
    }
}

static void TrySomeLoops()
{
    int count = 0;

    while (count < 10)
    {
        count = count + 1;
    }

    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }

    Console.WriteLine("The answer is " + count);
}
```

What does the TryAnIf method write to the console?

.....

.....

What does the TryAnIfElse method write to the console?

.....

What does the TrySomeLoops method write to the console?

.....

We didn't include answers for this exercise in the book. After you write down the answers, create a new console app—just like you did in Chapter 1—and add this code, then check the output to see if your answers are correct.

Game Design...and Beyond

Mechanics

The **mechanics** of a game are the aspects of the game that make up the actual gameplay: its rules, the actions that the player can take, and the way the game behaves in response to them.

- Let's start with a classic video game. The **mechanics of Pac-Man** include how the joystick controls the player on the screen, the number of points for dots and power pellets, how ghosts move, how long they turn blue and how their behavior changes after the player eats a power pellet, when the player gets extra lives, how the ghosts slow down as they go through the tunnel—*all* of the rules that drive the game.

  If you haven't played Pac-Man before, take a few minutes and watch a video of its gameplay. Even better, play it yourself! There are many ways to play it. One easy way is to search Google for "Pac-Man"—they created a playable Pac-Man game for its 30th anniversary that works in your browser using the arrow keys.

- When game designers talk about a **mechanic** (in the singular), they're often referring to a single mode of interaction or control, like a double jump in a platformer or shields that can only take a certain number of hits in a shooter. It's often useful to isolate a single mechanic for testing and improvement.
- **Tabletop games** give us a really good way to understand the concept of mechanics. Random number generators like dice, spinners, and cards are great examples of specific mechanics.
- You've already seen a great example of a mechanic: the **timer** that you added to your animal matching game changed the entire experience. Timers, obstacles, enemies, maps, races, points…these are all mechanics.
- Different mechanics **combine** in different ways, and that can have a big impact on how the players experience the game. Monopoly is a great example of a game that combines two different random number generators—dice and cards—to make a more interesting and subtle game.
- Game mechanics also include the way the **data is structured and the design of the code** that handles that data—even if the mechanic is unintentional! Pac-Man's legendary *level 256 glitch*, where a bug in the code fills half the screen with garbage and makes the game unplayable, is part of the mechanics of the game.
- When we talk about the mechanics of a game written in C#, **that includes the classes and the code**, because they drive the way that the game works.

Here's another one of those "Game Design...and Beyond" sections. This one is all about mechanics, an important part of video game design. If you're not interested in writing video games, read these sections anyway! They have important concepts that we'll build on throughout the book.

I bet the concept of mechanics can help me with **any kind of project**, not just games.

Definitely! Every program has its own kind of mechanics.

There are mechanics at every level of software design. They're easier to talk about and understand in the context of video games. We'll take advantage of that to help give you a deeper understanding of mechanics, which is valuable for designing and building any kind of project.

The mechanics of a game determine how hard or easy it is to play. Here's an example: make Pac-Man move faster or slow down the ghosts, and the game gets easier. That doesn't necessarily make it better or worse—just different. And guess what? The same idea applies to how you design your classes! You can think of ***how you design your methods and fields*** as the mechanics of the class. The choices you make about how to break up your code into methods or when to use fields make them easier or more difficult to use.

Bullet Points

- **Methods** are made up of statements. Calling a method executes its statements in order.
- Putting statements into a method and giving it a name helps make your code **easier to read**.
- When a method runs out of statements or executes a **`return`** statement, **execution resumes after the statement** that originally called the method.
- A variable's **type** determines what kind of data—like whole or decimal numbers, text, or true/false values—that it can hold
- You need to **assign values** to variables before you can use them.
- **Operators** like +, -, *, and / perform manipulations on the data stored in variables. The = operator assigns a value, while the == operator compares two values.
- **`if` statements** tell your program to do certain things only when the conditions you set up are (or aren't) true.
- **Loops** execute a set of statements over and over again until a condition is met. **`for`**, **`while`**, and **`do`**/**`while`** loops all iterate over statements multiple times, but they work differently from each other.
- Visual Studio's **code snippets** feature helps you write `if` statements and loops.

The Sens-AI elements are all about helping you use an AI chatbot as a virtual C# sensei. We wanted to start the first one with an example of just how powerful AI chatbots can be, so we asked Copilot to write this introduction. (We also asked it for permission to include its output in the book.)

Exploring C# with AI Chatbots: A Unique Learning Journey

Welcome to the fascinating world of C# development! As an author, you've already embarked on a mission to make learning engaging and memorable. Now, let's take it up a notch by introducing a powerful ally: **AI chatbots**.

"As an author"? Something seems weird here.

Why AI Chatbots?

Imagine having a friendly, tireless companion by your side—one that not only answers your questions but also challenges you to think critically, nudges you in the right direction, and celebrates your victories. That's precisely what AI chatbots can do for aspiring C# developers.

In this section, we'll delve into the pedagogical uses of AI chatbots, exploring how they can enhance the learning experience, spark curiosity, and empower learners to explore C# independently.

The Generative Magic Behind Chatbots

Before we dive in, let's demystify the magic. Designers harness a variety of machine learning and deep learning techniques to create large language models (LLMs) that form the backbone of generative AI chatbots. These bots can generate text, simulate conversations, and adapt to user input—while maintaining an engaging and conversational tone.

Seven Approaches to Educational Chatbot Use

- **AI as Mentor:** Timely feedback is crucial for learning. An AI chatbot can provide students with frequent, immediate, and adaptive feedback. Whether it's essay structure or code debugging, the chatbot acts as a mentor, guiding learners toward mastery.
- **AI as Tutor:** Effective tutoring involves personalized instruction and skill-building. Chatbots can supplement traditional tutoring, offering explanations, analogies, and open-ended questions tailored to individual interests.
- **AI as Collaborator:** Imagine brainstorming with an AI buddy. Chatbots can collaborate on problem-solving, code design, and creative thinking. They're the perfect coding companions for late-night sessions.
- **AI as Debugger:** Debugging code can be frustrating. Chatbots can analyze code snippets, identify errors, and suggest fixes. They're like virtual debugging partners, always ready to lend a hand.
- **AI as Code Generator:** Need inspiration? Chatbots can generate code snippets, templates, and even entire functions. They're the Swiss Army knives of C# development.
- **AI as Quizmaster:** Reinforce learning with quizzes. Chatbots can quiz learners on syntax, concepts, and best practices. Plus, they won't judge you for forgetting semicolons.
- **AI as Curator:** Curate resources, recommend tutorials, and keep learners updated. Chatbots can be your personalized C# library, accessible 24/7.

Your Hands-On Journey Begins

Throughout this section, we'll guide you through hands-on tasks. You'll reflect on educational uses of AI chatbots, access commonly used generative AI tools, and practice interacting with a chatbot. Remember, while AI is powerful, critical examination is essential. Let's explore, learn, and code together!

We agree!

"Sensei" is a Japanese term that encompasses the role of teacher or instructor.

Sens-AI

First lesson: AI chatbots aren't perfect...and that's OK

When you use an AI chatbot like Copilot, ChatGPT, or Gemini, you start with a **prompt**, or a question, statement, or other kind of query to initiate a conversation. We used this prompt to generate the introduction to this section:

> *I'm the author of Head First C#, a book that aims to help people learn about development with C# in a unique and engaging way. A section of that book is about how to use AI chatbots as a tool to learn and explore C# on their own. Can you help me write the introduction to that section?*

It came up with a great introduction—it's good enough for us to use in this book! But it isn't perfect. Take a look at the second sentence that starts, "As an author, you've..."—that's not quite right. The chatbot may have gotten a bit confused because we started our prompt with "I'm the author." This book isn't for authors, it's for people learning to be C# developers. *Your* mission isn't making learning engaging and memorable—that's *our* mission!

It's pretty amazing that the AI gave us such great text, even if we might want to edit it a little (like replace 'pedagogical' with 'teaching-related').

How to use AI *safely*: Trust but verify

Using AI chatbots is a skill that's becoming increasingly important to developers. One of our goals is to help you use AI chatbots in a way that's **safe**, which means that you have to *confirm two things* ***yourself***: that everything you learn from an AI is actually and factually **correct**, and that any code the AI generates for you ***does what you want it to do***. We'll include these Sens-AI elements throughout this to help you learn how to confirm those things and use AI safely.

Here's an example of the strengths and weaknesses of AI. We gave this prompt to Copilot, ChatGPT, and Gemini:

```
I have the following for loop in C#:

int p = 2;
for (int q = 2; q < 32; q = q * 2)
{
  while (p < q)
  {
   // How many times will the next
   // statement be executed?
   p = p * 2;
  }
  q = p - q;
}

How does it work, and how many times are the inner and outer loops executed?
```

We copied this code directly from the "Sharpen your pencil" earlier in this chapter. You can include code directly in your prompts.

When you're typing a prompt into a browser-based AI chatbot, you may need to use *Shift-Enter to add line breaks*. We recommend using a text editor like VSCode, Windows Notepad, or macOS TextEdit to edit your prompts. You can copy and paste them into the browser—and then you can save them in a folder for later.

The AI chatbots all gave an excellent explanation of how the loop worked—and all three **gave us *wrong* answers** for how many times the loops are executed**.** In fact, they all gave different answers. We asked the same question a few days later, and all three chatbots gave *three entirely different wrong answers*. And that's OK! AI technology is always improving. It may never be perfect, but it will always be a **helpful and valuable learning tool**.

AI researchers are constantly updating their models with questions and their correct answers, so it may get the answer right because the question and answer are in its database.

Try it out yourself

Open Copilot (*https://copilot.microsoft.com*), ChatGPT (*https://chat.openai.com*), Gemini (*https://gemini.google.com)*, or any other AI chatbot, and type in the prompt about loops that we just showed you. Modify the code in the prompt to replace it with other loops in the "Sharpen" exercise. Did the AI explain the loops well? Did it get the counts right?

AI chatbots don't always give correct answers, in part because they use statistical models—*not* actual thinking!—to generate sentences. But if you understand their limitations and practice using them, they can be really useful.

Controls drive the mechanics of your user interfaces

In Chapter 1, you built a game using Button and Label **controls**. There are a lot of different ways that you can use controls, and the choices you make about what controls to use can really change your app. Does that sound weird? It's actually ***very similar to the way we make choices about mechanics in game design***. If you're designing a tabletop game that needs a random number generator, you can choose to use dice, a spinner, or cards. If you're designing a platformer, you can choose to have your player jump, double jump, wall jump, or fly (or do different things at different times). The same goes for apps: if you're designing an app where the user needs to enter a number, you can choose from different controls to let them do that…***and that choice affects how your user experiences the app***.

Meet some of the controls you'll use in this book

Most of the chapters in this book feature a .NET MAUI project. We included them so you can go beyond console apps and start learning how to build visual apps. In those projects, you'll use many different controls to build each app's **user interface** (or **UI**)—or the way the window is laid out so the user can interact with it—of each app.

Here are the controls you've seen so far.

An *Image control* does exactly what you'd expect it to do—it displays an image. In this case, it's displaying the image in a file called *dotnet_bot.png.*

Hello, World!

A *label* displays text. You can set the font size, color, spacing, and text decorations (like italics or boldface).

A *Button control* shows a clickable button. It can call a method when you click it, and you can set or change its text.

We'll spend the rest of this chapter building a MAUI app to experiment with controls. You can find an equivalent web development project in the *Head First C# Blazor Learner's Guide*, which you can download from our GitHub page: *https://github.com/head-first-csharp/fifth-edition*

The user interface, or UI, is the part of the app that your user interacts with. In a console app, the UI is made up of text, and the user uses the keyboard to interact with it. In a MAUI app, the UI is built using controls.

Other controls you'll use in this book

Controls are common user interface components, and they serve as the building blocks of your UI. The choices you make about what controls to use change the mechanics of your app.

Most of the chapters in this book contain a .NET MAUI project. You'll use various controls to build the UI for each of those apps. Here are a few of the ones that you'll use.

We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our apps—not just games.

This is a multiline Label control.

A *Label* can include multiple lines, which are separated by line breaks so it knows how to split them up.

Enter some text

An *Entry control* lets your user enter text. It displays a placeholder, or lighter-colored text that gives the user some information about what they should type.

Off On

A *Switch* is a horizontal button that lets the user toggle (or switch back and forth) between two states, in this case on and off.

- +

These are two different controls that let users enter numbers. A *Stepper* (on the left) presents the user with two buttons to increment or decrement—add or remove one—to a value. A *Slider* (on the right) lets the user slide back and forth to choose a decimal number.

Pick a bird

Pigeon

Duck
Pigeon
Penguin
Ostrich
Owl

A *Picker* lets the user choose an item from a list. It looks a little different in Windows (on the left) and macOS (on the right), but both versions function in exactly the same way.

Pigeon

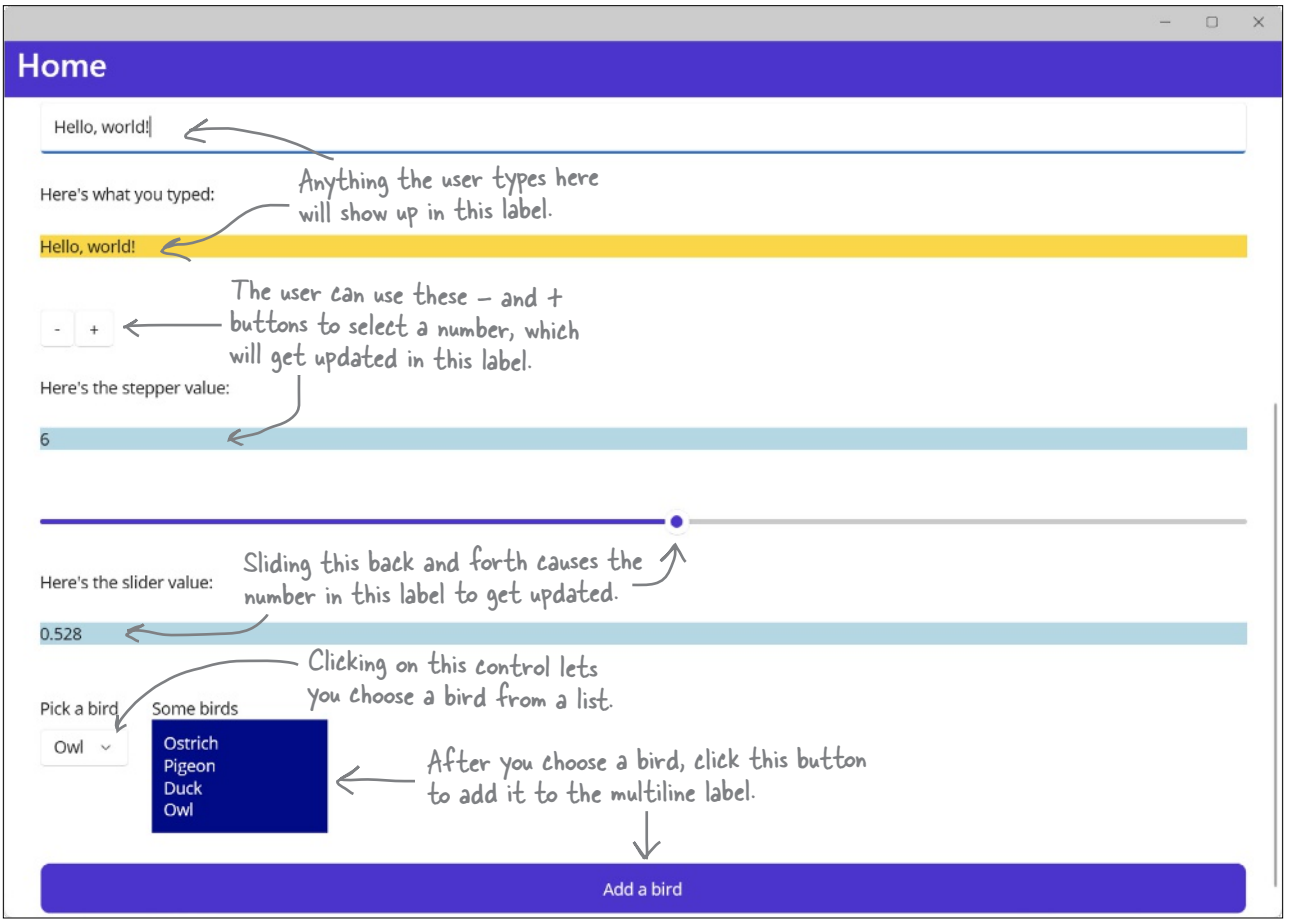
Pick a bird

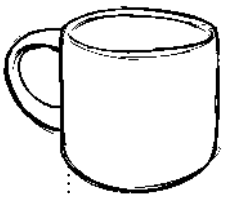
Duck
Pigeon
Penguin
Ostrich
Owl

Done

Build a .NET MAUI app to experiment with controls

You've probably seen most of the controls we just showed you (even if you didn't know all of their official names). Now let's **create a .NET MAUI app** to get some practice using some of them. The app will be really simple—the user will use controls to enter values, and the app will display those values.





Relax

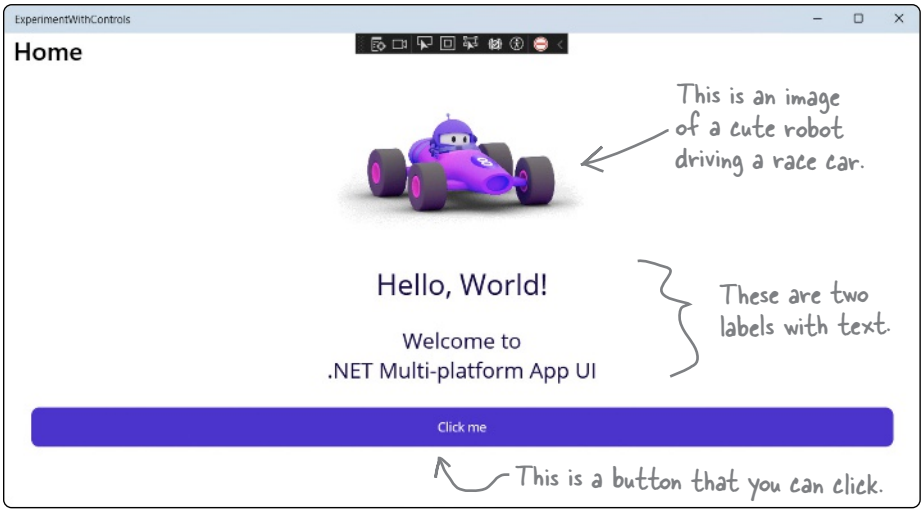
Don't worry about memorizing the XAML in this project. You'll pick it up throughout the book.

This Do this! *and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.*

Create a new app to experiment with controls

Do this!

Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project ***ExperimentWithControls***. Run your new .NET MAUI app. It will pop up a window with a picture of a cute robot, text that says *Hello, World!*, and smaller text that says *Welcome to .NET Multiplatform APP UI*, and finally a button with the label *Click me*.



Editing XAML and expanding or collapsing tags in VSCode works just like Visual Studio.

Now go back to Visual Studio and **double-click the file *MainPage.xaml*** to open it. Use the expand/collapse buttons in the left margin to collapse the `<Image>`, `<Label>`, and `<Button>` tags. Each of those tags corresponds to one of the controls in your app.

```
1   <?xml version="1.0" encoding="utf-8" ?>
2   <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
3                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4                x:Class="ExperimentWithControls.MainPage">
5
6       <ScrollView>
7           <VerticalStackLayout
8               Padding="30,0"
9               Spacing="25">
10              <Image.../>
15
16              <Label.../>
20
21              <Label.../>
26
27              <Button.../>
33          </VerticalStackLayout>
34      </ScrollView>
35
36  </ContentPage>
```

This <Image> tag displays the picture of the robot in an Image control.

Click here to collapse or expand a tag.

These two <Label> tags create Label controls that display the two lines of text.

This <Button> tag adds the Button control to the page.

The XAML for your MAUI page starts with a `ContentPage` tag, which can contain a single control—in this case, it's a *ScrollView*, which scrolls its content and displays a scrollbar on the side.

A ScrollView contains a single control. Yours contains a *VerticalStackLayout*, which can contain multiple controls (like the Image, two Labels, and Button on your page) and displays them stacked on top of each other vertically.

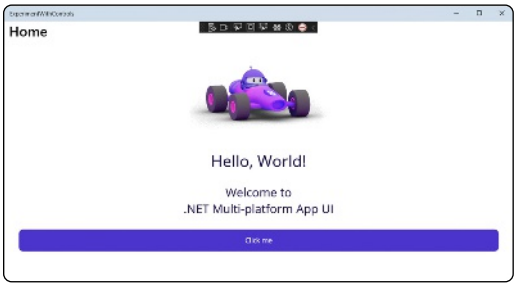
Explore your new MAUI app and figure out how it works

When you created your new .NET MAUI app, Visual Studio used a **template** to create the files for your app, substituting the name that you specified (*ExperimentWithControls*) in various lines in the files. Let's dig into the project that you created.

Do this!

1 **Create a new .NET MAUI project called ExperimentWithControls.**
Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project *ExperimentWithControls*.

2 **Run your app and click on the button.**
When you run the app, you'll see the app window—it should look like this:



MAUI is cross-platform, which means you'll see the same app—with the same code!—whether you're running Windows or macOS.

You know you want to click the "Click me" button. Go ahead! The label on the button will change from "Click me" to "Clicked 1 time" and increment (or add one) every time you click the button.

If your app doesn't pause on the breakpoint, make sure you're starting it with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu

3 **Investigate how the counter on the button works.**
Go to the Solution Explorer, expand *MainPage.xaml*, and **open *MainPage.xaml.cs***. Find the line that has the statement `count++;` and **place a breakpoint** on it.

```
0 references
12  private void OnCounterClicked(object sender, EventArgs e)
13  {
14      count++;
15
16      if (count == 1)
17          CounterBtn.Text = $"Clicked {count} time";
18      else
19          CounterBtn.Text = $"Clicked {count} times";
20
21      SemanticScreenReader.Announce(CounterBtn.Text);
22  }
```

The C# code for a page in your MAUI app is called *code-behind*. The XAML code and the C# code in the code-behind file work together to make your page work.

Before you go to the next step, *read the code*. Can you figure out how the button works?

4 Click on the button and step through the code.

Add a watch for the count variable, just like you did earlier in the chapter. Then use **"Step Over" (F10)** to step through the code. Here's what the OnCounterClicked event handler method does:

- First it executes `count++` to increment (or add one to) the `count` variable.
- Next it uses an `if` statement to check if the `count` variable equals 1. If it does, then it sets the button's text to "Clicked 1 time."
- If it doesn't equal 1, it sets the button's text to "Clicked {count} times"—you'll learn more about exactly what the $ dollar sign and {brackets} do in Chapter 5 (it's called *string interpolation*).

5 Click on the button and step through the code.

The program should pause on the breakpoint, just like you saw earlier in the chapter:

```
12  private void OnCounterClicked(object sender, EventArgs e)
13  {
14      count++;
15
16      if (count == 1)
17          CounterBtn.Text = $"Clicked {count} time";
18      else
19          CounterBtn.Text = $"Clicked {count} times";
20
21      SemanticScreenReader.Announce(CounterBtn.Text);
22  }
```

This "if" statement sets the text on the Count button to "time" if count equals 1, or "times" if it has any other value.

Add a watch for the `count` variable, just like you did earlier with the OperatorExamples console app. It should start out with the value 0. Press the Step Over button or F10 to go to the next statement.

Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
count	0	int

The count variable starts with the value 0. After the "count++" statement is executed, the new value is 1. Each time you click the button, count++ increases the value of the count variable by 1.

Keep stepping through the code. The `if` statement checks whether the count value is equal to 1. If it is, it executes this statement, which updates the text on the button:

```
CounterBtn.Text = $"Clicked {count} time";
```

Go back to the window with the XAML code. Find this line:

```
<Button
    x:Name="CounterBtn"
```

This is the x:Name property. It gave the button the name "CounterBtn" that you can use in your C# code.

Every control can have a name. The x:Name property sets the name of the control—in this case, the button is named CounterBtn—and your C# code can use that name to make the control do things.

The XAML for Your Button Up Close

You've been editing the XAML code in your *MainPage.xaml* file—are you starting to get comfortable with it? This is a great time to take a closer look at the part of your XAML that displays the button.

Here's the **`Button`** tag. Take a look at each of its five properties. Can you figure out what they all do?

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />
```

The x:Name property gives your control a name you can use in your code.

The first property is x:Name, which sets the name of the control so you can use it in your C# code:

```
x:Name="CounterBtn"
```

You just saw a control name in action. When you clicked the button, the event handler method executed this statement to set the button's text, using the name CounterBtn set by the x:Name property:

```
CounterBtn.Text = $"Clicked {count} time";
```

This line uses the CounterBtn name to update the text displayed on the button.

The Text property determines the text that's displayed on the button.

The XAML for a Buttons control has lots of options to change its appearance—like this next property:

```
Text="Click me"
```

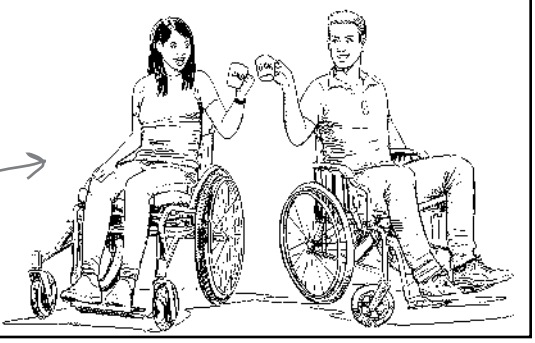
The button displays "Click me" when you first run the app. That line of code in the method changes the text to "Clicked 1 time" the first time you click it, then "Clicked 2 times" when you click it again. That line of code starts with the name of the control (CounterBtn), followed by a period, followed by Text, the name of the property.

SemanticProperties help you make your apps *accessible.*

When we create our apps, we want everyone to be able to use them—and that includes people with disabilities.

A **screen reader** is a tool that lets people who are blind, visually impaired, or have learning disabilities or other conditions that interfere with their ability to read use our visual apps. Semantic properties help your app work with a screen reader.

A screen reader is an accessibility tool for people with visual, learning, or other disabilities—just like a wheelchair is an accessibility tool for people with mobility-related disabilities. They're both really important for helping to make everyday things more accessible to everyone.



The XAML for Your Button Up Close

Use a screen reader to experiment with the SemanticProperites.Hint property.

The best way to make your apps accessible is to use them the way someone with accessibility issues would—in this case, using a screen reader built into your operating system.

- **In Windows, start the Narrator app.** You can run it from the Start menu, or use Windows logo key + Ctrl + Enter to turn Narrator on or off, and Windows logo key + Ctrl + N to bring up Narrator settings. Narrator will display a window with an overview of how Narrator works. It will also start to read the contents of that window, displaying a box around the section of the window that it's reading. You can go back to that window to turn off Narrator.
- **In MacOS, start the VoiceOver utility.** It lives in the Applications/Utilities folder, but if your keyboard has Touch ID, the easiest way to turn it on or off is to press and hold the Command key while you quickly press Touch ID three times. By default the VoiceOver utility displays a welcome dialog—press the V key or click the Use Voiceover button to start VoiceOver.

Once you have Narrator or VoiceOver running, switch to your app window. You'll hear a voice telling you details about what's on the screen. People with visual impairments often have trouble using a mouse, so they use the keyboard to interact with apps—and you'll do the same thing. **Press the Tab key** to navigate to the Click Here button. The screen reader will announce that you are on a button. Listen closely—you'll hear it speak the SemanticProperties.Hint value: "Counts the number of times you click."

Press Enter to click the button. Your app will execute code that includes this statement:

```
SemanticScreenReader.Announce(CounterBtn.Text);
```

When it does, the screen reader will announce the contents of the button ("Clicked 1 time").

The Clicked property tells your app what event handler method to run when the button is clicked.

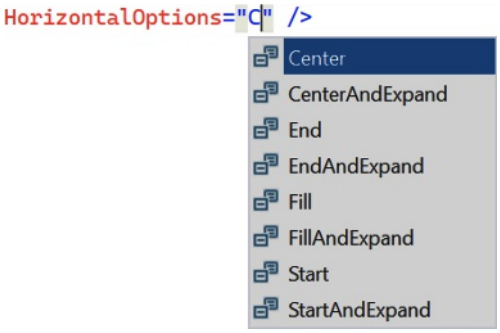
Take a look at the next property in the button's XAML code:

```
Clicked="OnCounterClicked"
```

When you click the button, your app uses that property to figure out which method to run. You saw this in action when you placed a breakpoint on the first line of that method.

The HorizontalOptions property centers your button.

When you run your app, the Click Me button is centered in the middle of the window. Go back to the code editor, select the word `Center` in that line of XAML code, and type **C**. Visual Studio will display an IntelliSense pop-up with all of the different options. Try selecting Start or End, then run your app again—now the button will be displayed on the left or right side of the window. Experiment with all of the different horizontal options for Button control.



Add an Entry control to your app

An **Entry control** displays a box for the user to enter text. You'll add one to your app, and you'll use a really useful tool in Visual Studio to do it: the **Toolbox window**. The Toolbox is a feature of Visual Studio that makes it easy to add controls to your app:

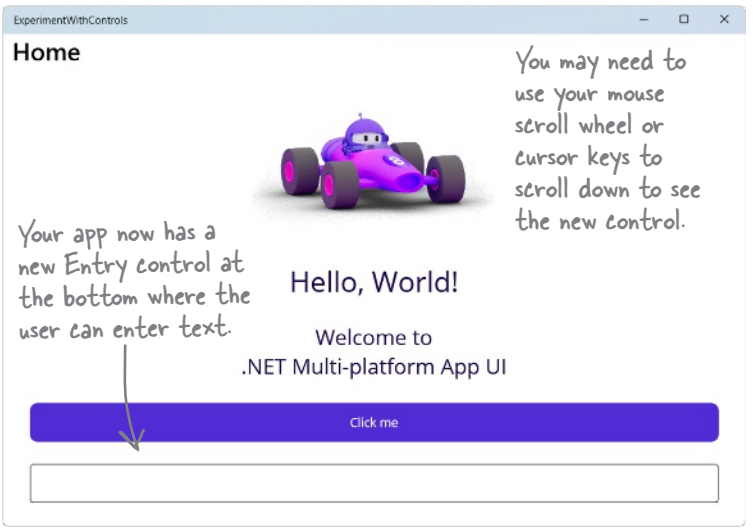
1. Stop your app, then open the MainPage.xaml editor window in Visual Studio.
2. Place your mouse cursor just after the closing /> bracket at the end of your Button control tag, then **press Enter three times** to add three blank lines. Click on the second line that you just added, so there's a blank line above your mouse cursor and another blank line below it.
3. Open the Toolbox window in Visual Studio (if it isn't already open) by **choosing Toolbox from the View menu**.
4. **Double-click Entry** in the Toolbox window. Visual Studio will automatically add an `<Entry>` tag at your cursor location, on that blank line you added.

If you have trouble with the Toolbox, you also can ***type the XAML code*** *directly into the editor.*

Here's what you should see in your XAML code:

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Fill" />

<Entry Placeholder="" />
```

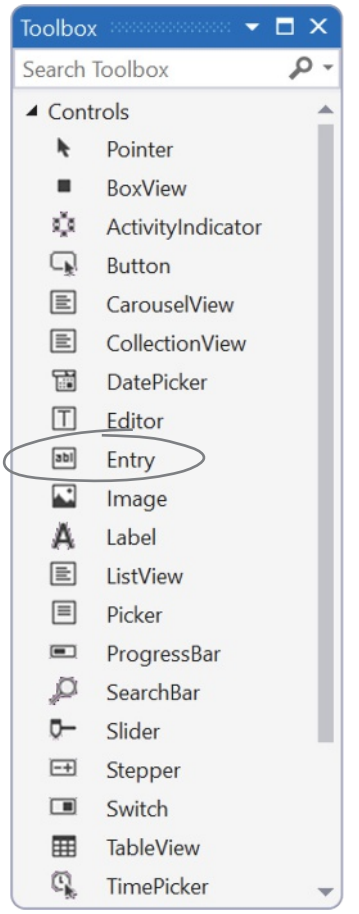
Now run your app. Congratulations, you just added a control for entering text!



You may need to use your mouse scroll wheel or cursor keys to scroll down to see the new control.

Your app now has a new Entry control at the bottom where the user can enter text.

Visual Studio's Toolbox window helps you add new controls to your XAML code. If you don't see the Toolbox window, choose "Toolbox" from the View menu to display it.



If you're using VSCode, it doesn't have a Toolbox, so you'll need to type the XAML in by hand.

The screenshot and XAML for the app are from the MAUI template that's part of .NET 8. If you're using a different version of .NET, you may see a different image and slightly different XAML.

Add properties to your Entry control

Let's make your Entry control a little more usable by adding **placeholder text**, or text that appears in a lighter color to help the user understand what they're supposed to enter.

Edit the XAML code for your Entry control to **add a Placeholder property**. And since we always want our apps to be accessible to people who use screen readers, **add a SemanticProperties.Hint property** too. Notice that when you add the properties, they show up in Visual Studio's typeahead pop-up window, making it easier for you to add them.

Your Entry tag should look like this:

```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text" />
```

Now run your app—you'll see a new Entry control at the bottom. The placeholder text will appear as ("Enter some text" in a lighter color, and will disappear as soon as you type text into it.

If you don't see the control at the bottom of the window, use the cursor keys or mouse scroll wheel to scroll down to it..

At the time we're writing this, VSCode does not have a Toolbox, so you'll need to type all of the XAML by hand. Make sure it matches the XAML in the book exactly, or the app won't work.

there are no Dumb Questions

Q: Why did the Entry control get added to the bottom of my app? How did it know where in the window to display?

A: When you created a new .NET MAUI app, Visual Studio used a template that generated the XAML code for the main page in the *MainPage.xaml* file. This file contains a set of **nested tags**, or tags that contain other tags—so one tag's start and end appear after the start and before the end of another tag. Each of these tags *creates a specific kind of control* that determines how the page is displayed.

The outermost tag in your app's XAML is a `<ContentPage>` opening tag, which defines a single view that contains the rest of the page. If you scroll down to the bottom of the file, you'll see the closing `</ContentPage>` tag. Right inside that `<ContentPage>` is a `<ScrollView>` tag—everything between the opening `<ScrollView>` and closing `</ScrollView>` tags defines contents that will automatically display a scrollbar that lets you scroll up and down if it's too long for the page. The `<ScrollView>` tag contains a `<VerticalStackLayout>` tag, with a matching `</VerticalStackLayout>` closing tag at the bottom. A VerticalStackLayout can contain a series of controls, one after another. Each of those controls will be displayed on the page in a vertical stack, in the order that they appear in the file.

So since the Entry control is at the bottom of the file just above the closing `</VerticalStackLayout>` tag, it will appear at the bottom of the page. And because it's nested inside the `<ScrollView>...</ScrollView>` tags, if you make your window shorter than the height of the page, you'll be able to scroll down to it.

Make your Entry control update a Label control

Your app already has two Label controls. Let's add a third one and make it display everything the Entry does, so when you enter or update text in the Entry it automatically updates the Label.

1. **Use the Toolbox to add a new Label control to the bottom of your page.**
When you drag the Label out of the Toolbox, it will have an empty Text property:

```
<Label Text="" />
```

Change the Text property to make it display text. Then give it a SemanticProperties.Description property. This is what will get read aloud if your user is using a screen reader:

```
<Label Text="Here's what you typed:"
     SemanticProperties.Description="Here's what you typed:" />
```

You can add line breaks between properties to make them easier to read.

2. **Use the Toolbox to add a *second* Label control under the one you just added.**
Every time the user changes the text in your Entry control, the app will update this new Label to show the text that they typed. Drag a new Label control out of the Toolbox and drop it in your XAML code between the Label control that you just added and the closing `</VerticalStackLayout>` tag. Then set its properties:

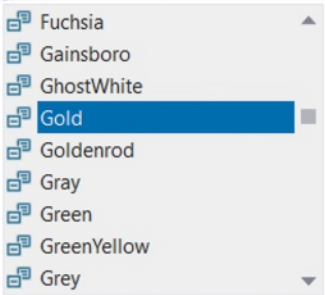
- You'll be writing code to set the Label text, so delete the Text property.
- Since you're going to write code that updates the Label, you'll need to give it a name. Use an x:Name property to name it EnteredText:
`x:Name="EnteredText"`
- Keep making your app accessible by adding a description for people using a screen reader:
`SemanticProperties.Description="The text that the user entered"`

Your new Label should look like this:

```
<Label x:Name="EnteredText"
     SemanticProperties.Description="The text that the user entered" />
```

3. **Give your label a background color.**
Add a BackgroundColor property. When you start typing, Visual Studio will pop up an IntelliSense window. Choose Gold for the background color.

```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
    BackgroundColor=""/>
```



You can use Visual Studio's IntelliSense to help you add properties. Once you add it, you'll see a box with a preview of the color in the XAML editor.

```
BackgroundColor="Gold"/>
```

4 Use the Properties window to add a bottom margin.

So far you've been adding properties by writing XAML code by hand. Luckily, Visual Studio has some useful tools to help you edit your XAML. The Properties window gives you an easy way to edit the properties on your controls.

Click the XAML for your Label control so the cursor is somewhere between the tags. Go to the Properties window (if you don't see it, use the View menu to display it) and find Margin. Enter 20 for the lower margin to give it a 20 pixel margin (where a pixel is 1/96th of an inch on an unscaled screen).

When this square turns black (or circle if you're using macOS), it means that the property has a value set. You can click it to see a menu.

Enter 20 for the bottom margin to give 20 units of space between the label and the bottom of the page.

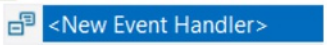
Your property should now look like this:

```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
     BackgroundColor="Gold" Margin="0,0,0,20"/>
```

5 Add an event handler method.

Back in Chapter 1, you used event handler methods so your animal matching game could respond to mouse clicks and timer ticks. Now you know more about C# methods—this is a good chance to apply that knowledge by creating a new event handler method that updates the EnteredText control when the user types in the Entry control. Add a TextChanged property to your Entry control. When it comes time to enter the value, Visual Studio will suggest the value ***<New Event Handler>***:

```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text"
    TextChanged=""
    />
```



Don't forget—VSCode may not have this feature yet, so you might need to type the Entry_TextChanged method in by hand.

Press Return or use the mouse to accept the suggestion—this will cause Visual Studio to **add a new event handler method called Entry_TextChanged automatically**. You probably also noticed that it also displayed this message when you were adding the event handler:

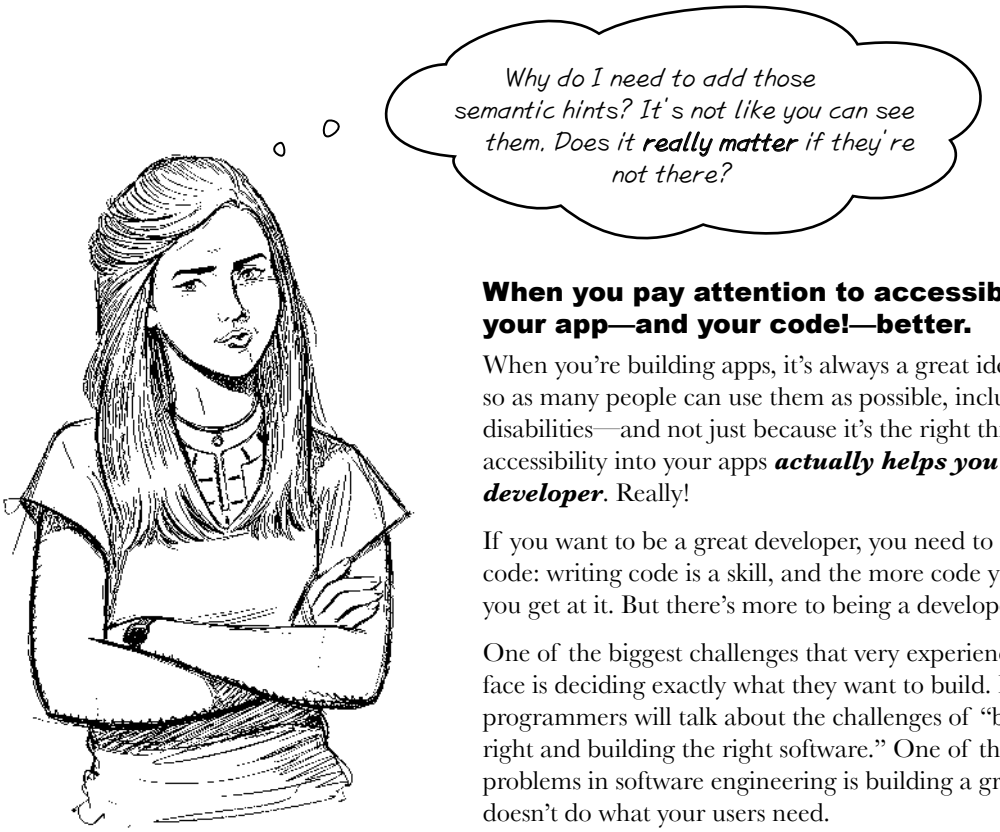
> Bind event to a newly created method called 'Entry_TextChanged'. Use 'Go To Definition' to navigate to the newly created method.

Right-click on Entry_TextChanged and choose Go To Definition. This will open up *MainPage.xaml.cs* and jump directly to the method that Visual Studio added. Add this line of code to the method:

```
private void Entry_TextChanged(object sender, TextChangedEventArgs e)
{
      EnteredText.Text = e.NewTextValue;
}
```

If you're using VSCode, open MainPage.xaml.cs and add this just above the last closing bracket.

Now **run your app**. You should see a label that says "Here's what you typed:" followed by a gold-colored label. Click on the Entry control and type some text—it will appear in the gold-colored label immediately.



When you pay attention to accessibility, it makes your app—and your code!—better.

When you're building apps, it's always a great idea to create them so as many people can use them as possible, including people with disabilities—and not just because it's the right thing to do. Building accessibility into your apps ***actually helps you become a better developer***. Really!

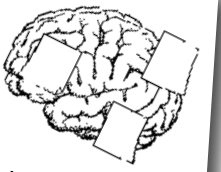
If you want to be a great developer, you need to get practice writing code: writing code is a skill, and the more code you write, the better you get at it. But there's more to being a developer than "just" coding.

One of the biggest challenges that very experienced developers face is deciding exactly what they want to build. In fact, a lot of programmers will talk about the challenges of "building the software right and building the right software." One of the most common problems in software engineering is building a great product that doesn't do what your users need.

That's where accessibility can help you. Building accessible code well means taking the time to **really understand** how people with disabilities will use your app. Taking the time to understand and empathize with them will help you build your app better—and it's great practice for skills that will help you build the right software.

One of our big goals with this book is to help you learn important skills that will help you become an all-around great developer. Understanding your users is a really important skill, and paying attention to accessibility is a great way to get better at it!

Make it Stick



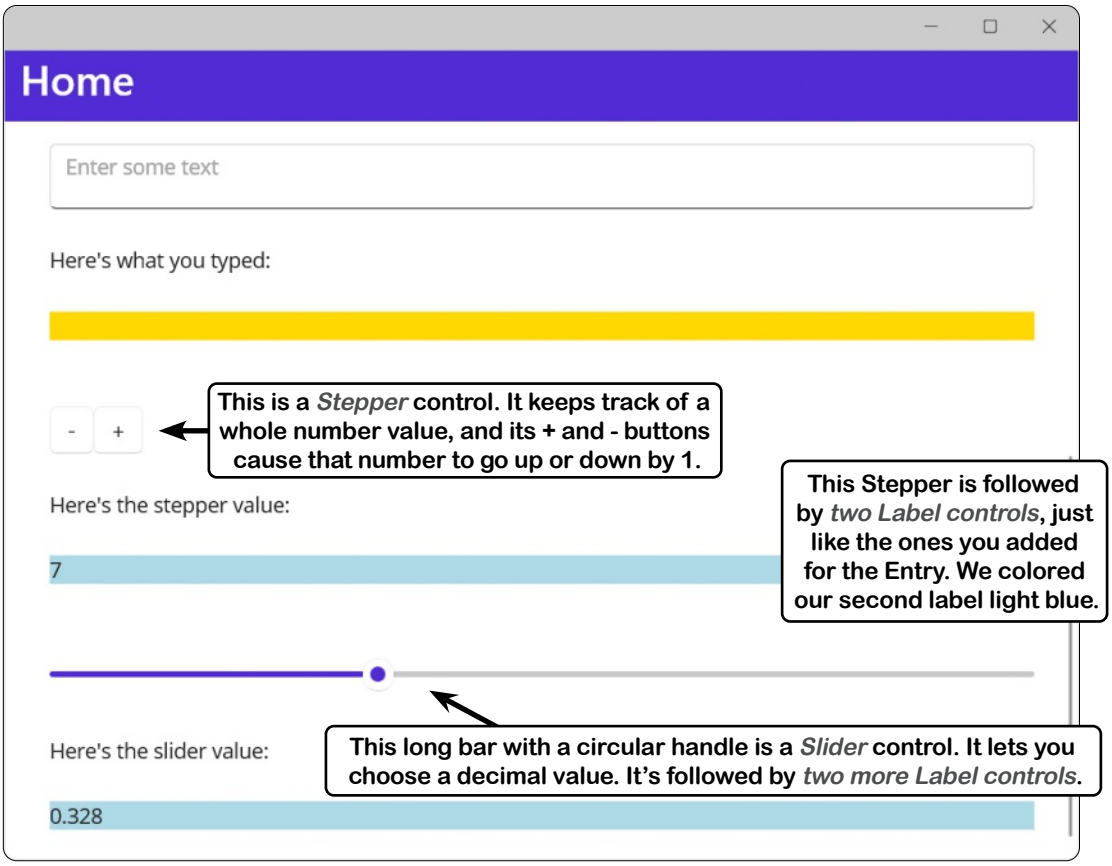
Here's a great way to get accessibility ideas to stick in your brain—especially if you don't have a disability. Turn on your screen reader, then **leave it on** while you code or do other work. Once you're used to it, **close your eyes** and keep working. Can you work using just the screen reader?

Using a screen reader is an effective way to get a better feel for accessibility.

Don't forget, it's absolutely fine to peek at the solution—especially if you're using VSCode and trying to figure out how to add the event handler methods.

Exercise

You added Entry and Label controls to your app—and Visual Studio's Toolbox window, Properties window, and IntelliSense helped you. Can you add six more controls to your app to let your user enter numeric values?



Use the Toolbox window, Properties window, and Visual Studio editor to **add a Stepper control, two Label controls, a Slider control, and two more Label controls** to your app (if you're using VSCode, add them manually).

The two Label controls that display the values should have the BackgroundColor property set to LightBlue. **Name them *StepperValue* and *SliderValue***. Make sure you **add SematicProperties.Description properties**.

You want your app to automatically update the StepperValue control every time the stepper value changes, so **add a ValueChanged event handler** to the Stepper control. Add this line of code to the event handler:

```
StepperValue.Text = e.NewValue.ToString();
```

Then add a ValueChanged event handler **to the Slider control**. It should be identical, except that it updates the SliderValue label instead of the StepperValue label. Don't forget to **add SemanticDescription.Hint properties** to your Stepper and Slider controls.

If you're using VSCode, you'll need to add the Stepper_ValueChanged and Slider_ValueChanged methods manually. They're just like Entry_TextChanged, just replace "Text" with "Value" in the first line. Visual Studio's Toolbox automatically adds these properties to the Stepper tag: Minimum="0" Maximum="10" Increment="1"—and these to the Slider tag: Minimum="0". Maximum="1".

Exercise Solution

Here's the XAML to add the six controls to *MainPage.xaml*:

```
    <Label x:Name="EnteredText"
        SemanticProperties.Description="The text that the user entered"
        BackgroundColor="Gold" Margin="0,0,0,20"/>

    <Stepper Minimum="0" Maximum="10" Increment="1"
                SemanticProperties.Description="Lets you enter a whole number"
                ValueChanged="Stepper_ValueChanged" />

    <Label
        Text="Here's the stepper value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="StepperValue"
        SemanticProperties.Description="The number the user chose with the Stepper"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>


    <Slider Minimum="0" Maximum="1" ValueChanged="Slider_ValueChanged" />

    <Label
        Text="Here's the slider value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="SliderValue"
        SemanticProperties.Description="The number the user chose with the Slider"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>

</VerticalStackLayout>
```

This is the Label control that was already in your XAML code—make sure you put your six new controls below it.

These are the default properties when you drag the Stepper out of the Toolbox. Try experimenting with them.

You can add this ValueChanged property just like you did with TextChanged on your Entry control.

Here's the Slider control. It has the default properties, plus a ValueChanged property.

Here's the Label that displays the Slider value. It works exactly like the Label you used to show the value in the Entry control.

This is the closing VerticalStackLayout tag that was already in your XAML code—make sure you put your six new controls above it.

Here are the event handler methods to add to MainPage.xaml.cs:

```
private void Stepper_ValueChanged(object sender, ValueChangedEventArgs e)
{
    StepperValue.Text = e.NewValue.ToString();
}

private void Slider_ValueChanged(object sender, ValueChangedEventArgs e)
{
    SliderValue.Text = e.NewValue.ToString();v
}
```

The two event handlers for the Stepper and Slider controls update the Label.

In the exercise instructions, we gave you this line of code:

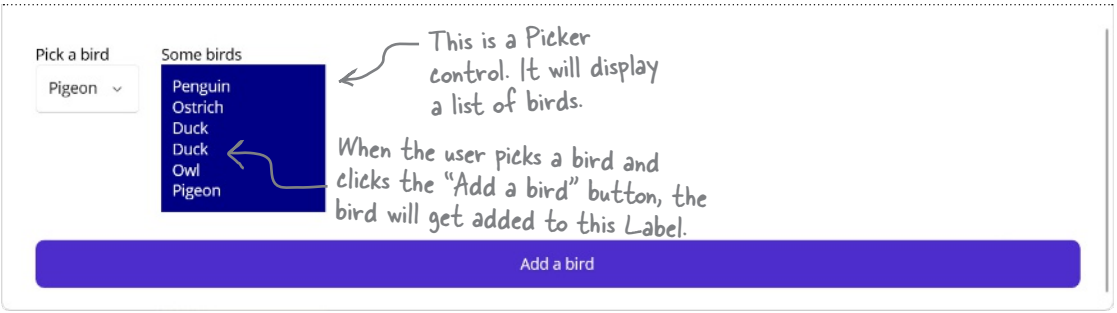
```
SliderValue.Text = e.NewValue.ToString();
```

What do you think `.ToString()` does?

Here's a hint: Stepper and Slider controls can only provide numeric values, but Labels can only display text.

Combine horizontal and vertical stack layouts

In this last part of the exercise, you'll add a **Picker control**, which displays a list of items that you can pick from. You'll also use a Label control to display the values that were picked. Here's what it will look like:



Notice how the Label and Picker controls are next to each other? You'll get that layout by using a **HorizontalStackLayout control**. It works just like the VerticalStackLayout control, which causes all of the controls you've added to your app so far to be stacked vertically on top of each other, except instead they get stacked horizontally next to each other.

You'll nest one Layout inside another

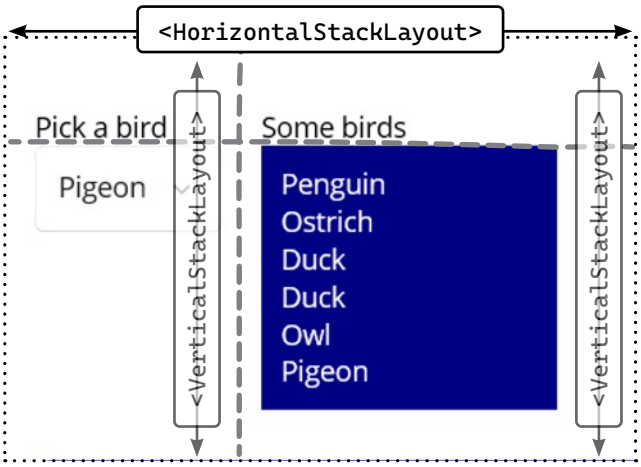
We'll use **nesting**—where one layout control lives inside another one—to create a more complex layout.

Here's how it will work:

This HorizontalStackLayout will get nested inside the outer VerticalStackLayout that's used to lay out the entire page.

```
<HorizontalStackLayout>

   <VerticalStackLayout>
     <Label Text="Pick a bird" ... />
     <Picker x:Name="BirdPicker" ... />
   </VerticalStackLayout>

   <VerticalStackLayout>
      <Label Text="Some Birds" ... />
      <Label x:Name="Birds" ... />
   </VerticalStackLayout>

</HorizontalStackLayout>
```

You'll use this HorizontalStackLayout to position two VerticalStackLayout controls next to each other.



This nested VerticalStackLayout contains a Label that says "Pick a bird" and a Picker control named BirdPicker, stacked vertically.

This nested VerticalStackLayout contains a Label that says "Some birds" and a Label named Birds that will get updated when the user clicks the button, also stacked vertically.

Add a Picker control to display a list of choices

A **Picker control** displays a list of items in a dropdown so the user can pick one of them. Let's add one to your app.

1 **Add the XAML for a Picker control and a Label for it to update.**

You've already seen how a VerticalStackLayout control lets you stack controls on top of each other. You can also stack controls horizontally by adding a **HorizontalStackLayout control**.

Go ahead and **add this XAML code** just above the closing </VerticalStackLayout> tag. You can type it all or use the Toolbox. When you add the Clicked event for the button, **press Tab** to let Visual Studio generate an event handler method for you, just like you did earlier (add the method manually in VSCode).

```
<HorizontalStackLayout Spacing="20">

    <VerticalStackLayout>
        <Label Text="Pick a bird" SemanticProperties.Description="Picks a bird"/>
        <Picker x:Name="BirdPicker" />
    </VerticalStackLayout>

    <VerticalStackLayout>

        <Label Text="Some birds"
               SemanticProperties.Description="A list of birds"/>

        <Label x:Name="Birds"
               Padding="10" MinimumWidthRequest="150"
               TextColor="White" BackgroundColor="DarkBlue"
               SemanticProperties.Description="Shows the added birds" />

    </VerticalStackLayout>

</HorizontalStackLayout>

<Button x:Name="AddBird" Text="Add a bird" Margin="0,0,0,20"
        SemanticProperties.Hint="Adds a bird"/>
```

This <Button ... /> tag should be just above the closing </VerticalStackLayout> tag that's already in MainPage.xaml.

2 **Initialize the Picker with a list of birds.**

Open the MainPage.xaml.cs file and find the MainPage method at the top. This method gets run every time the page loads. Insert two lines after `InitializeComponent();` and **add this code**.

```
public MainPage()
{
    InitializeComponent();

    BirdPicker.ItemsSource = new string[] {
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    };
}
```

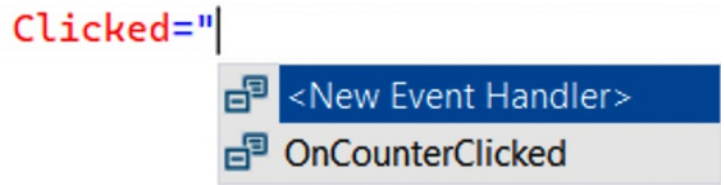
You used the x:Name property to name your Picker control "BirdPicker"—this sets the list of items in the picker that get displayed when the user clicks on it.

Put your code at the end of the MainPage method.

Open the MainPage.xaml.cs file and add this code to the MainPage method. Be careful with the square and curly braces, quotes, and commas.

3 **Fill in the event handler for the Button control.**

The XAML we gave you for the Button control didn't include a Clicked event handler. Use Visual Studio's IntelliSense pop-up to add a new event handler to the C# code, just like you did with the Entry control:



Don't forget—VSCode may not have this feature yet, so you might need to type the AddBird_Clicked method in by hand.

Since you used the x:Name property to name your Picker control AddBird, Visual Studio created an empty event handler method called AddBird_Clicked:

```
private void AddBird_Clicked(object sender, EventArgs e)
{

}
```

If you're using VSCode, you might need to open MainPage.xaml.cs and add this entire method. Put it just above the bottom closing bracket in the file.

Add this line of code to the AddBird_Clicked method:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Take a closer look at the line of code—let's break down exactly what it does.

1. The line starts with `Birds.Text =` ... which means it's setting the text in the Bird label.
2. The text is being set to `Birds.Text +` followed by additional things—this means it's going to take whatever is in the Label and **append text** to it, or add additional text to the end.
3. The first thing that gets appended is `Environment.NewLine`, which adds a line break. The Label control will display **multiline text**, adding a line break every time it sees a line break.
4. After the line break, it appends `BirdPicker.SelectedItem`—this is the item that's currently selected in the Picker control.

4 **Run your app and use your new Picker control.**

Scroll to the bottom of the app, choose a bird from the Picker, and click the Add a bird button—it will get added to the Label that contains the birds. Select a few more birds and add them.

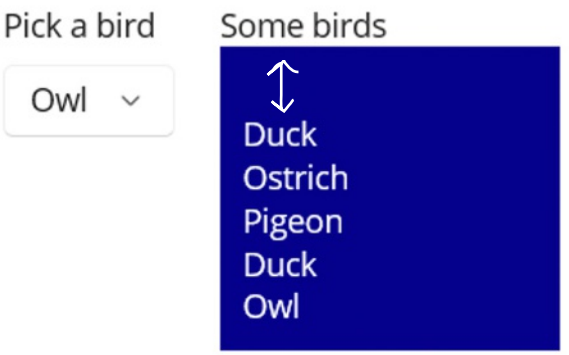
The Label expands every time you add a bird because it adds a new line break. If the button scrolls off the bottom of the page, just scroll down to it.

Hold on. My app doesn't match the first screenshot that you showed us. It looks like ***there's SOME EXTRA SPACE*** *at the top of the label! The code has a bug.*

You're right! The app doesn't match the screenshot.

Take a look at the screenshot we showed you earlier:

Run your app and try adding those same birds. When you get to the first owl, you'll see extra space at the top of the label:



Oops! It looks like we've got some extra space at the top of the label that shows the birds that you picked.

Looks like we've got a bug. Time to put on your Sherlock Holmes cap.
Let's sleuth out this bug!

Sleuth it Out

The Case of the Extraneous Space

Understanding a bug is the first step in fixing it.

In Chapter 1, we looked at the code carefully and found several clues to help us solve the Case of the Unexpected Match. But as you keep going through this book, your apps will get longer and longer, and while looking at the code is a good start, it may not always be the best way to figure out what's causing a bug.

Luckily, the Visual Studio debugger is a great tool for that. (That's why it's called a debugger: it's a tool that helps you get rid of bugs!)

Reproduce the bug

It seems obvious that there's a problem. But as Sherlock Holmes once said, "There is nothing more deceptive than an obvious fact." When you're sleuthing out bugs, you can't just rely on what seems obvious. You need to confirm for yourself exactly what's going on. The way to do that is to **reproduce the bug**.

Stop your app. Make sure it's not running, so you've got a fresh start. Then do this:

1. Start your app again.
2. Pick Duck and click the "Add a bird" button.
3. Pick Ostrich and click the "Add a bird" button.
4. Pick Pigeon and click the "Add a bird" button.
5. Pick Duck and click the "Add a bird" button.
6. Pick Owl and click the "Add a bird" button.

Your app should now look exactly like the screenshot:

"There is nothing more deceptive than an obvious fact."
— Sherlock Holmes

Now restart your app, then try it again with different birds. You should still see extra space at the top of the label. You can make the bug happen over and over again, at will. That means the problem is **reproducible**: you can follow a set of steps to make it happen. Reproducing a bug is a great first step to fixing it.

Before you go on, can you sleuth out what's causing the extra space to get added?

Sleuth it Out

Every good investigation starts by identifying a list of suspects

When you're tracking down a bug, what's the first thing you should do? You could start placing breakpoints in the code…but where? **The first step in debugging is thinking.** Look at your code, think about how it works, and try to imagine where the bug might be. That will help you figure out where to put your breakpoints.

So let's think through the code. It starts with a button—and the button calls a method:

```
<Button x:Name="AddBird" Clicked="AddBird_Clicked" Text="Add a bird"
        Margin="0,0,0,20" SemanticProperties.Hint="Adds a bird"/>
```

All of the code to add the bird to the label is in that **AddBird_Clicked method**. Now we have a suspect!

Remember, If your app doesn't pause on the breakpoint, make sure you're starting the app with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu.

IDE Tip: Using the debugger

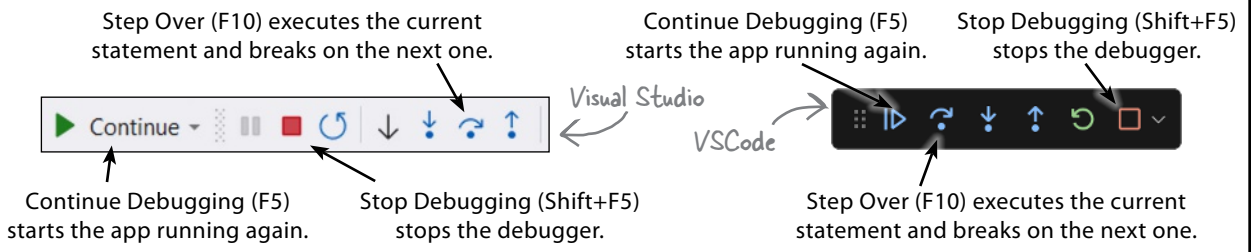
You're going to be using the debugger a lot in this book! We've walked you through it a few times, but as you get further in the book and write more and more code, you should feel comfortable using the debugger on your own.

Let's start with **a few tips** to help you get comfortable debugging your code:

- Think before you debug. Read through your code. Understand how it works (and not just how *you think it works*).
- Use the Watch window, Locals window, and hovering over variables to keep track of their values. They all do the same thing—show you the value of a variable—so you can decide which one you feel most comfortable with.
- Don't be afraid to restart your app. Stop and start your code frequently—every time you run your code, you're *running an experiment*. Run it as many times as it takes to understand what's going on.

Here's a handy **list of useful debugger commands**. They may feel strange at first, but they'll be second nature soon:

- When you press the triangle Run button in the toolbar or choose Start Debugging (F5), Visual Studio starts running your code in the debugger. You can place a breakpoint whether or not the debugger is running.
- To place a breakpoint, click on a line of code and choose Toggle Breakpoint (F9) from the Debug menu.
- When your code hits a breakpoint, it stops running so you can inspect variables.
- When Visual Studio breaks on a breakpoint, the toolbar shows you the commands you can use to keep executing. Debugging code can be a little weird to get used to if you haven't done it before, so try sticking to just these four commands—here's where you'll find them in the IDE's toolbar, along with their keyboard shortcuts:



Sleuth it Out

Add a breakpoint and start debugging the code

Now that we have a suspect, let's catch it in the act. Add a breakpoint to the line in the AddBird_Clicked method:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Now **run your code**. Pick a bird, then click the "Add a bird" button. The debugger stops on your breakpoint. Next, **add a watch for Birds.Text**, just like you did earlier in the chapter. The value should be **`null`**:

Name	Value	Type
Birds.Text	null View	string

Then **step over that line of code** (F10) to run it. You should see this value:

Name	Value	Type
Birds.Text	"\r\nPigeon" View	string

The value of Birds.Text is a string: `\r\n` followed by the bird you picked. What do you think `\r\n` does?

NOTE: If you're using macOS, you'll see `\n` instead of `\r\n`.

Continue debugging (F5) to start your app running again. Pick a different bird and step over the line of code. Now have a look at the Birds.Text watch:

Name	Value	Type
Birds.Text	"\r\nPenguin\r\nOstrich" View	string

Repeat the process a few more times: continue debugging, pick a bird, click the button, step over, check the watch. Eventually your Birds.Text value will look something like this (you'll see `\n` instead of `\r\n` on macOS):

"`\r\n`Penguin`\r\n`Ostrich`\r\n`Pigeon`\r\n`Duck`\r\n`Owl`\r\n`Pigeon`\r\n`Duck`\r\n`Owl"

You've probably figured out by now that **the `\r\n` or `\n` is the line break**. The first time the AddBird_Clicked method is called, the Label text is empty (that's what the **`null`** value means), so when the app adds the current value **`(empty)`** plus a line break plus the bird, it adds an extra line break at the start of the string.

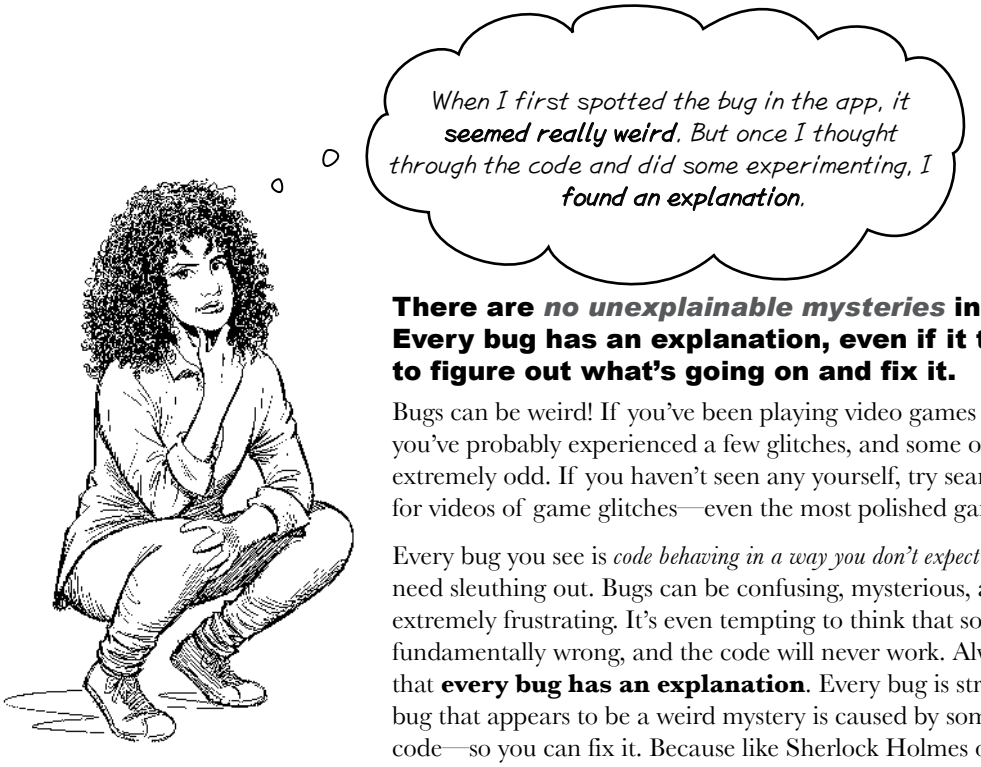
Now that we've found the culprit, we can fix the app. **Replace the AddBird_Clicked** method with this code, which uses a special method, **String.IsNullOrEmpty**, which checks if a string is empty:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(Birds.Text))
    {
        Birds.Text = Birds.Text + Environment.NewLine;
    }
    Birds.Text += BirdPicker.SelectedItem;
}
```

String.IsNullOrEmpty(Birds.Text) checks the value of Birds.Text and *returns* true if it's empty or false if it's not. The ! in front of it reverses that value, so the line break is only added if Birds.Text is empty. You'll learn all about how a method can return a value in the next chapter.

Run your app again and add a few birds to the label—there's no more empty space above it. **Your app is fixed!**

The Picker keeps growing each time you add a bird. Can you figure out which property gives it a maximum height?



There are *no unexplainable mysteries* in your code. Every bug has an explanation, even if it takes work to figure out what's going on and fix it.

Bugs can be weird! If you've been playing video games for a long time, you've probably experienced a few glitches, and some of them can be extremely odd. If you haven't seen any yourself, try searching the web for videos of game glitches—even the most polished game has bugs.

Every bug you see is *code behaving in a way you don't expect*. That's why bugs need sleuthing out. Bugs can be confusing, mysterious, and sometimes extremely frustrating. It's even tempting to think that something is fundamentally wrong, and the code will never work. Always remember that **every bug has an explanation**. Every bug is strange, but even a bug that appears to be a weird mystery is caused by something in your code—so you can fix it. Because like Sherlock Holmes once said, "It is a mistake to confound strangeness with mystery."

Bullet Points

- You'll use many different **controls** to build your app's user interface (or UI). The UI is the part of the application that your user interacts with.
- The C# code for a page in a MAUI app is called **code-behind**. The XAML code and the C# code in the code-behind file work together to make the page work.
- The x:Name property gives your control a name you can use in your code.
- When you pay attention to **accessibility**, it makes your app—and your code!—better. **Semantic properties** help you make your apps accessible by providing descriptions and hints for people who use screen readers.
- In XAML you can have **nested controls**, or tags that contain other controls, so one control's start and end tag appear after the start tag and before the end of another tag.
- You can use nested **HorizontalStackLayout** and **VerticalStackLayout** controls to create more complex layouts.
- The first step in debugging is thinking: look at your code, think about how it works, and try to imagine where the bug might be.
- **Reproducing a bug** is an important tool that helps you fix it. When you're debugging, you're **running an experiment** every time you run your code. Run it as many times as it takes to understand what's going on.

Unity Lab #1

Explore C# with Unity

Welcome to your first **Head First C# Unity Lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**. Unity will be a really valuable tool for that.

Unity is a cross-platform game development tool that you can use to make professional-quality games, simulations, and more. It's also a fun and satisfying way to **get practice with the C# tools and ideas** you'll learn throughout this book. We designed these short, targeted labs to **reinforce the concepts and techniques you just learned** to help you hone your C# skills. These labs are optional, but valuable practice... **even if you aren't planning to write games in C#**.

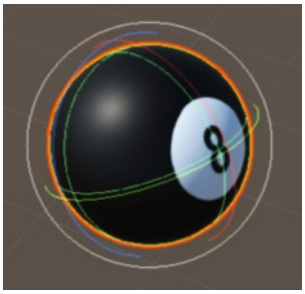
In this first lab, you'll get started with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes. That will lay down a foundation to write code in the next lab.

Unity is a powerful tool for game design

Welcome to the world of Unity, a complete system for designing professional-quality games—both two-dimensional (2D) and three-dimensional (3D)—as well as simulations, tools, and projects. Unity includes many powerful things, including...

A cross-platform game engine

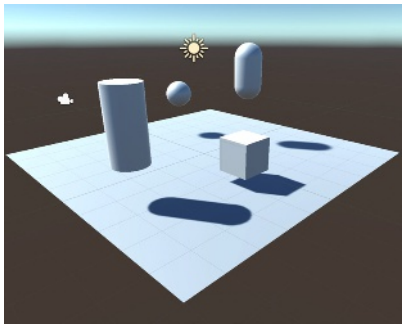
A **game engine** displays the graphics, keeps track of the 2D or 3D characters, detects when they hit each other, makes them act like real-world physical objects, and much, much more. Unity will do all of these things for the 3D games you build throughout this book.



A powerful 2D and 3D scene editor

You'll be spending a lot of time in the Unity editor. It lets you edit levels full of 2D or 3D objects, with tools that you can use to design complete worlds for your games. Unity games use C# to define their behavior, and the Unity editor integrates with Visual Studio to give you a seamless game development environment.

While these Unity Labs will concentrate on C# development in Unity, if you're a visual artist or designer, the Unity editor has many artist-friendly tools designed just for you. Check them out here: https://unity.com/solutions/artist-designers



An ecosystem for game creation

Beyond being an enormously powerful tool for creating games, Unity also features an ecosystem to help you build and learn. The Learn Unity page (*https://unity.com/learn*) has valuable self-guided learning resources, and the Unity forums (*https://forum.unity.com*) help you connect with other game designers and ask questions. The Unity Asset Store (*https://assetstore.unity.com*) provides free and paid assets like characters, shapes, and effects that you can use in your Unity projects.

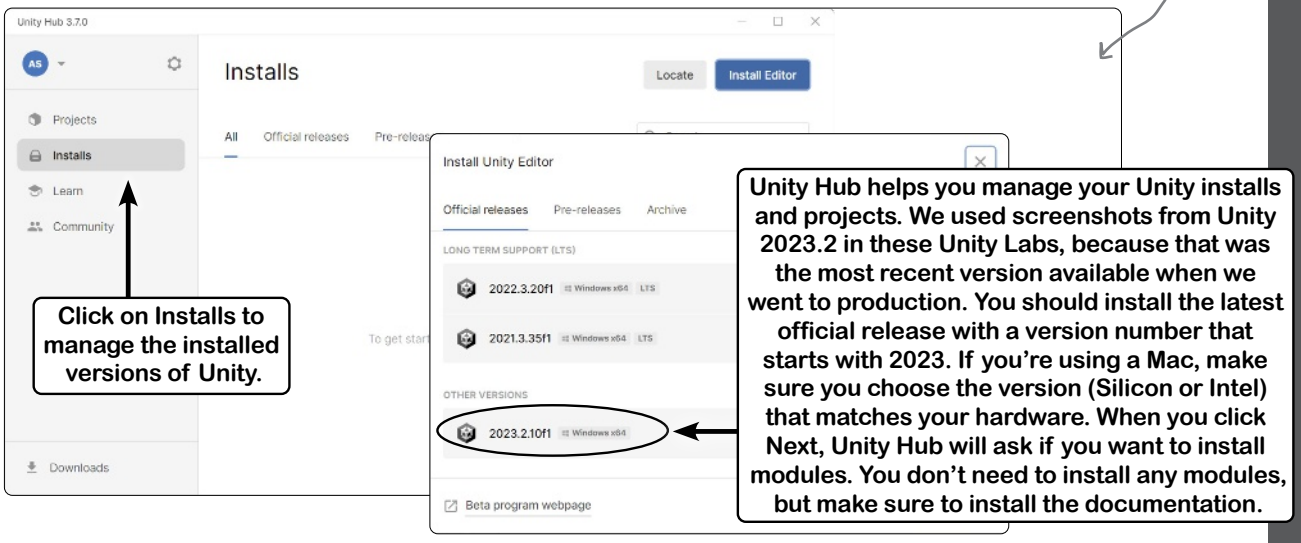
Our Unity Labs will focus on using Unity as a tool to explore C# and practicing with the C# tools and ideas that you've learned throughout the book.

The *Head First C#* Unity Labs are laser-focused on a **developer-centric learning path**. The goal of these labs is to help you ramp up on Unity quickly, with the same focus on brain-friendly just-in-time learning you'll see throughout *Head First C#* to ***give you lots of targeted, effective practice with C# ideas and techniques***.

Download Unity Hub

All of the screenshots in this book were taken with the free Personal Edition of Unity. You'll need to enter your unity.com username and password into Unity Hub to activate your license.

Unity Hub is an application that helps you manage your Unity projects and your Unity installations, and it's the starting point for creating your new Unity project. Start by downloading Unity Hub from *https://unity.com/developer-tools*—then install it and run it.



Unity Hub lets you install multiple versions of Unity on the same computer, so you should install the same version that we used to build these labs. **Click the Install Editor button** and install the latest official release that starts with ***Unity 2023***—that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

Unity Hub lets you have many Unity installs on the same computer. So even if there's a newer version of Unity available, you can use Unity Hub to install the version we used in the Unity Labs.

The Unity installer may prompt you to install a different version of Visual Studio. You can have multiple installations of Visual Studio on the same computer too, but if you already have one version of Visual Studio installed there's no need to make the Unity installer add another one.

You can learn more about installing Unity Hub on Windows and macOS here: *https://docs.unity3d.com/Manual/GettingStartedInstallingUnity.html*

Watch it!

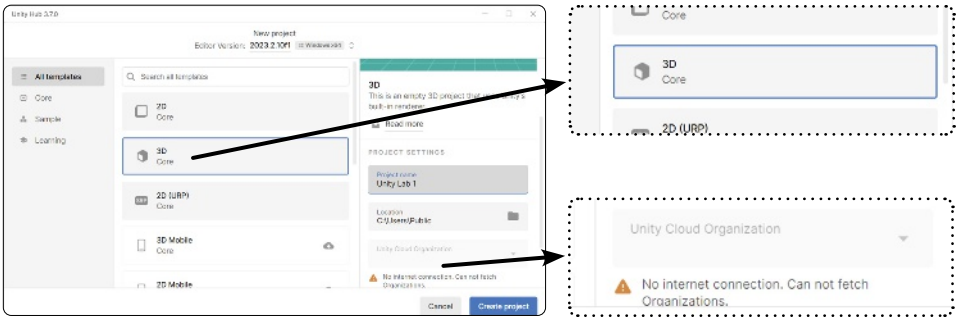
Unity Hub may look a little different.

*The screenshots in this book were taken with **Unity 2023.2 in dark mode** and **Unity Hub 3.7.0 in light mode**. You can use Unity Hub to install many different versions of Unity on the same computer, but you can only install the latest version of Unity Hub. The Unity development team is constantly improving Unity Hub and the Unity editor, so it's possible that what you see won't quite match what's shown on this page. The next version, **Unity 6**, is in early release as we go to print. Watch for PDFs of updated labs on our GitHub page: https://github.com/head-first-csharp/fifth-edition*

At the time we're writing this, Unity is planning on moving away from year-based versions and calling their next version Unity 6.

Use Unity Hub to create a new project

Click the New project button on the Project page in Unity Hub to create a new Unity project. Name it ***Unity Lab 1***, make sure the 3D template is selected, and check that you're creating it in a sensible location (usually the Unity Projects folder underneath your home directory).

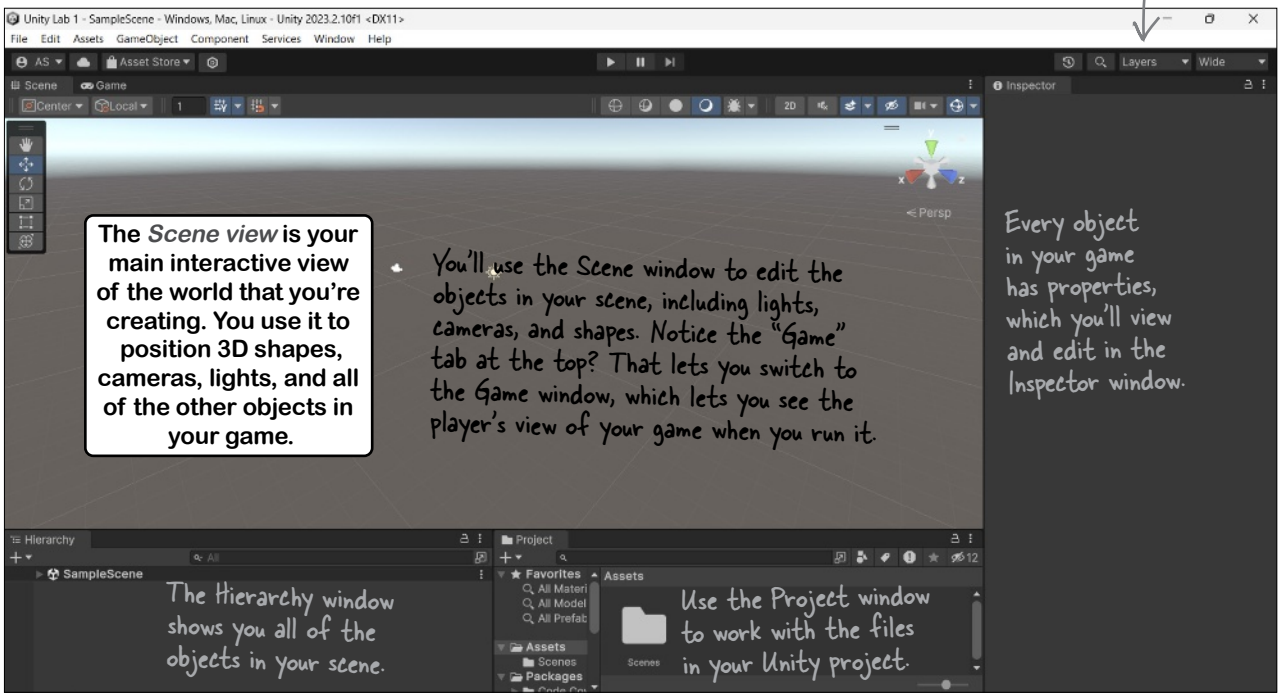


Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like Visual Studio does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

Work with your project in the Unity editor

Once your project is created, it will load in the **Unity editor**, a powerful tool that you'll use to create 3D environments. Here are some important parts of the Unity editor:

You can use this dropdown to change the layout of the Unity editor.



OK! You're all ready to get started on your first Unity project.

Take control of the Unity layout

The Unity editor is like an IDE for all of the parts of your Unity project that aren't C#. You'll use it to work with scenes, edit 3D shapes, create materials, and so much more.

When you started up Unity, did you notice that your screen looked a little different from our screenshot? Just like in Visual Studio, the windows and panels in the Unity editor can be rearranged in many different layouts. We chose a layout that works well for screenshots in a book. We also chose dark mode, which we think is easier to read when these pages are printed.

You can download PDFs of all of these Unity Labs and print them out if that makes it easier for you to follow along.

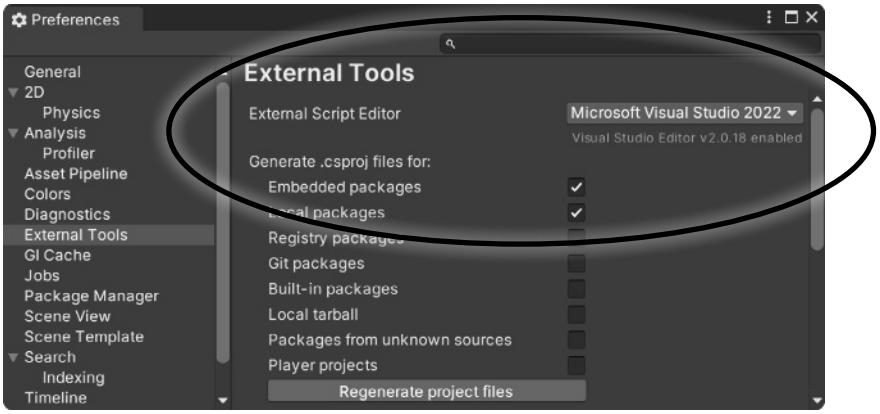
Choose the Wide layout to match our screenshots

We've chosen the Wide layout because it works well for the screenshots in these labs. Find the Layout dropdown in the toolbar and choose Wide so your Unity editor looks like ours.

Once you change the layout with the Layout dropdown on the right side of the toolbar, the dropdown changes its label to match the layout that you selected.

Set up Unity to work with Visual Studio

The goal of these Unity Labs is to give you an **exciting and fun way to explore C#**. The Unity editor works with Visual Studio and VSCode to make it easy to edit and debug the code for your games. Open the **Unity Preferences Window** (on Windows choose Preferences from the Edit menu; on a Mac choose Settings from the Unity menu). Click on External Tools on the left, click the External Script Editor dropdown, and **choose Visual Studio 2022** (or **Visual Studio Code** if you're using VSCode) from the list of options.



You can use Visual Studio to debug the code in your Unity games. Just choose Visual Studio as the external script editor in Unity's preferences.

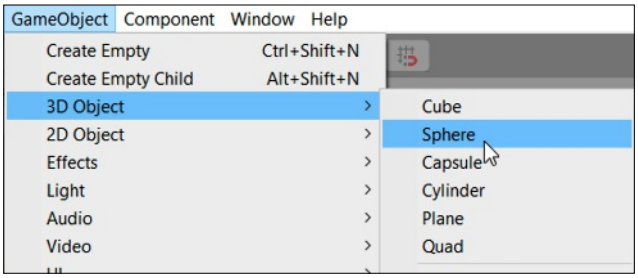
You'll write code and do some debugging with Visual Studio or VSCode in the next Unity Lab. This lab is about getting used to the way Unity works so you're ready to do that.

If you don't see Visual Studio in the External Script Editor dropdown, choose *Browse* and navigate to Visual Studio. On Windows it's normally an executable called devenv.exe in the folder C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\. On a Mac it's typically an app called Visual Studio in the Applications folder.

Your scene is a 3D environment

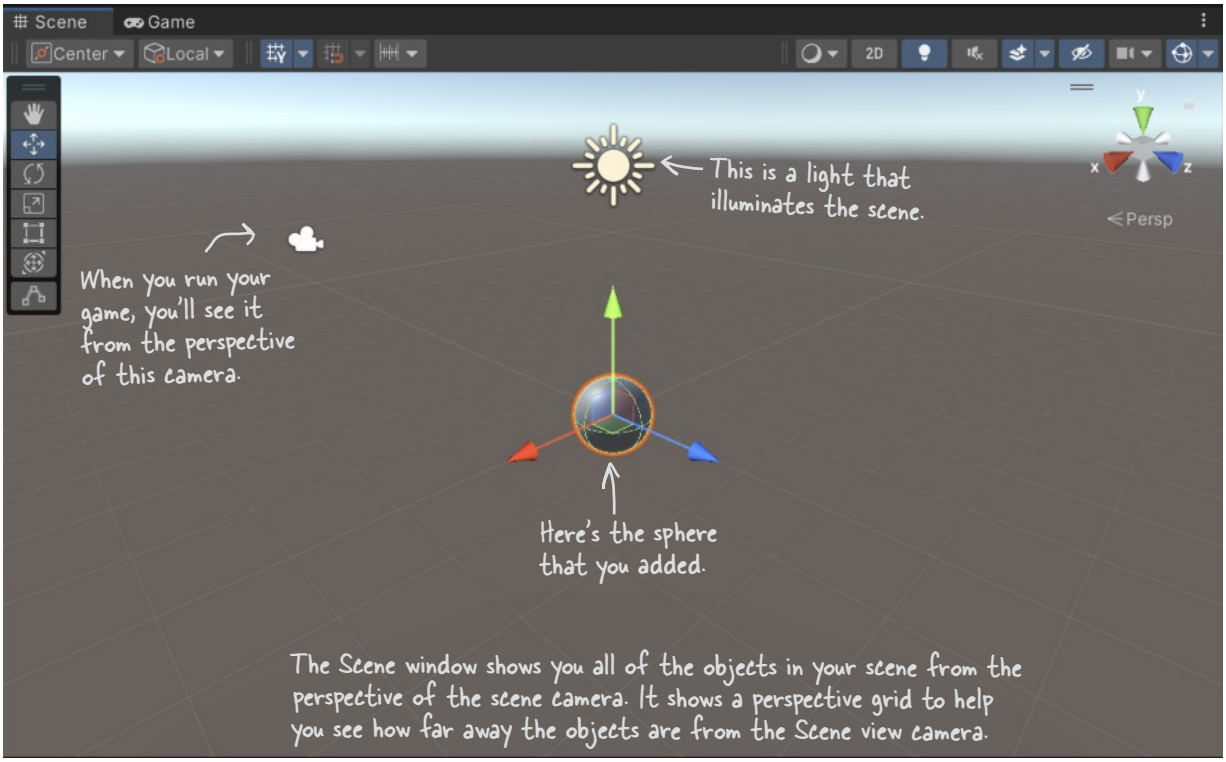
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project, Unity added a scene called SampleScene and stored it in a file called *SampleScene.unity*.

Add a sphere to your scene by choosing **GameObject >> 3D Object >> Sphere** from the menu:



These are called Unity's "primitive objects." We'll be using them a lot throughout these Unity Labs.

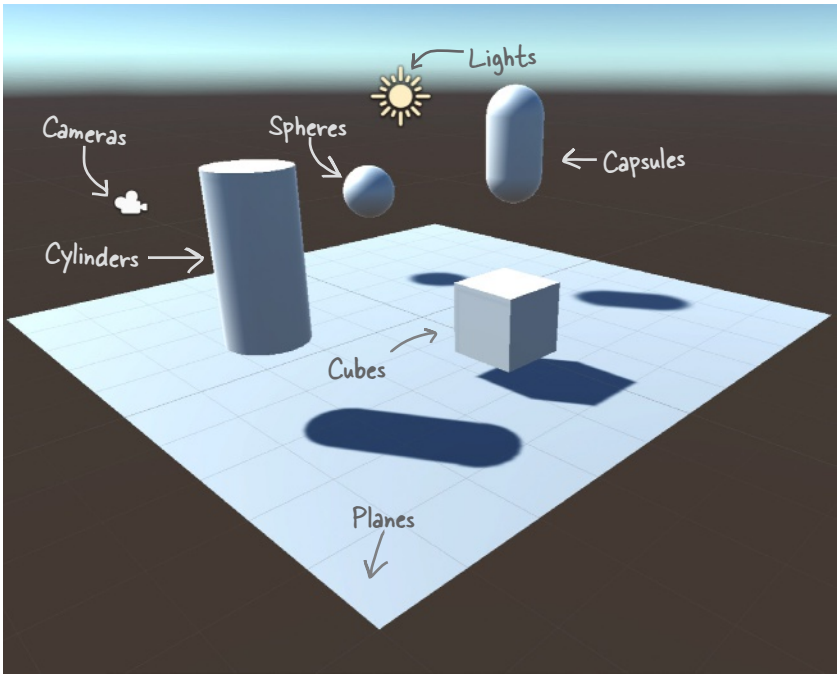
A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which "looks" at the scene and captures what it sees.



Unity games are made with GameObjects

When you added a sphere to your scene, you created a new **GameObject**. The GameObject is a fundamental concept in Unity. Every item, shape, character, light, camera, and special effect in your Unity game is a GameObject. Any scenery, characters, and props that you use in a game are represented by GameObjects.

In these Unity Labs, you'll build games with different kinds of GameObjects, including:



GameObjects are the fundamental objects in Unity, and components are the basic building blocks of their behavior. The Inspector window shows you details about each GameObject in your scene and its components.

Each GameObject contains several **components** that provide its shape, set its position, and give it all of its behavior. For example:

- *Transform components* determine the position and rotation of the GameObject.
- *Material components* change the way the GameObject is **rendered**—or how it's drawn by Unity—by changing the color, reflection, smoothness, and more.
- *Script components* use C# scripts to determine the GameObject's behavior.

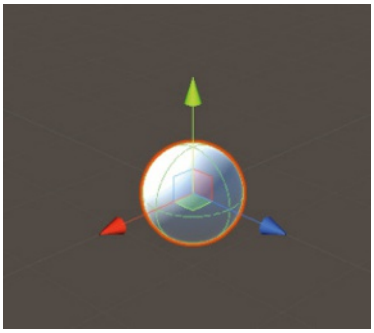
ren-der, verb.
to represent or depict artistically.
*Michelangelo **rendered** his favorite model with more detail than he used in any of his other drawings.*

Use the Move Gizmo to move your GameObjects

The Tools panel lets you choose **Transform tools**. If the Move Tool isn't selected, click on the sphere that you just added, then click the Move Tool in the **Tools overlay** to select it.

The *Tools overlay* lets you choose tools to manipulate GameObjects. You'll use the *Move Tool* to move your sphere around the scene. In the Wide view, the Tools overlay is vertical. You can right-click the two lines at the top to change its orientation so it's horizontal, or you can drag it to the toolbar or the side of the window to dock it.

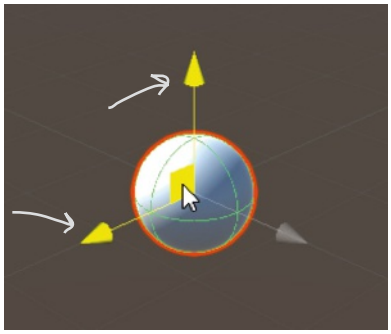
The **Move Tool** lets you use the **Move Gizmo** to move GameObjects around the 3D space. You should see red, green, and blue arrows and a cube appear in the middle of the window. This is the Move Gizmo, which you can use to move the selected object around the scene.



Using the Move Tool displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected. When you click the sphere and then choose the Move Tool, you'll see the Move Gizmo appear on the sphere. Click anywhere else in the scene to deselect the sphere and the Move Gizmo goes away.

Move your mouse cursor over the cube at the center of the Move Gizmo—notice how each of the faces of the cube lights up as you move your mouse cursor over it? Click on the *upper-left face* and drag the sphere around. You're moving the sphere in the X-Y plane.

When you click on the upper-left face of the cube in the middle of the Move Gizmo, its X and Y arrows light up and you can drag your sphere around the X-Y plane in your scene.



The Move Gizmo lets you move GameObjects along any axis or plane of the 3D space in your scene.

Move your sphere around the scene to get a feel for how the Move Gizmo works. Click and drag each of the three arrows to drag it along each plane individually. Try clicking on each of the faces of the cube in the Scene Gizmo to drag it around all three planes. Notice how the sphere gets smaller as it moves farther away from you—or really, the scene camera—and larger as it gets closer.

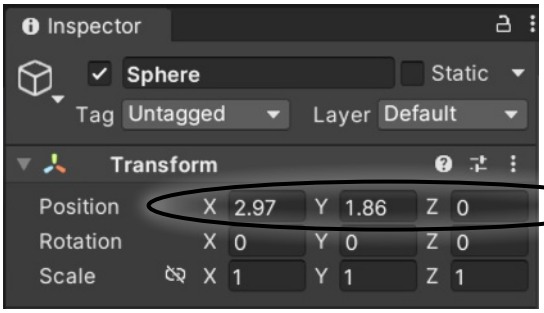
The Inspector shows your GameObject's components

As you move your sphere around the 3D space, watch the **Inspector window**, which is on the right side of the Unity editor if you're using the Wide layout. Look through the Inspector window—you'll see that your sphere has four components labeled Transform, Sphere (Mesh Filter), Mesh Renderer, and Sphere Collider.

Every GameObject has a set of components that provide the basic building blocks of its behavior, and every GameObject has a **Transform component** that drives its location, rotation, and scale.

You can see the Transform component in action as you use the Move Gizmo to drag the sphere around the X-Y plane. Watch the X and Y numbers in the Position row of the Transform component change as the sphere moves.

If you accidentally deselect a GameObject, just click on it again. If it's not visible in the scene, you can select it in the *Hierarchy window*, which shows all of the GameObjects in the scene. When you reset the layout to Wide, the Hierarchy window is in the lower-left corner of the Unity editor.

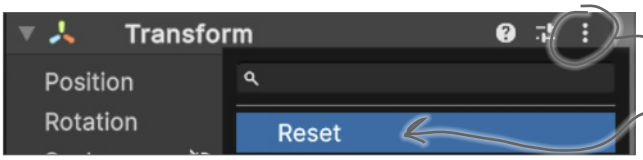


Did you notice the grid in your 3D space? As you're dragging the sphere around, *hold down the Control key*. That causes the GameObject that you're moving to snap to the grid. You'll see the numbers in the Transform component move by whole numbers instead of small decimal increments.

Try clicking on each of the other two faces of the Move Gizmo cube and dragging to move the sphere in the X-Z and Y-Z planes. Then click on the red, green, and blue arrows and drag the sphere along just the X, Y, or Z axis. You'll see the X, Y, and Z values in the Transform component change as you move the sphere.

Now **hold down Shift** to turn the cube in the middle of the Gizmo into a square. Click and drag on that square to move the sphere in the plane that's parallel to the Scene view camera.

Once you're done experimenting with the Move Gizmo, use the sphere's Transform component context menu to reset the component to its default values. Click the **context menu button** ( ) at the top of the Transform panel and choose Reset from the menu.



Use the context menu to reset a component. You can either click the three dots or right-click anywhere in the top line of the Transform panel in the Inspector window to bring up the context menu.

The position will reset back to [0, 0, 0].

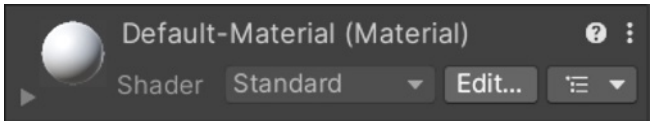
You can learn more about the tools and how to use them to position GameObjects in the Unity Manual. Click Help >> Unity Manual and search for the "Positioning GameObjects" page.

Save your scene *often*! Use File >> Save or Ctrl+S / ⌘S to save the scene right now.

Add a material to your Sphere GameObject

Unity uses **materials** to provide color, patterns, textures, and other visual effects. Your sphere looks pretty boring right now because it just has the default material, which causes the 3D object to be rendered in a plain, off-white color. Let's make it look like a billiard ball.

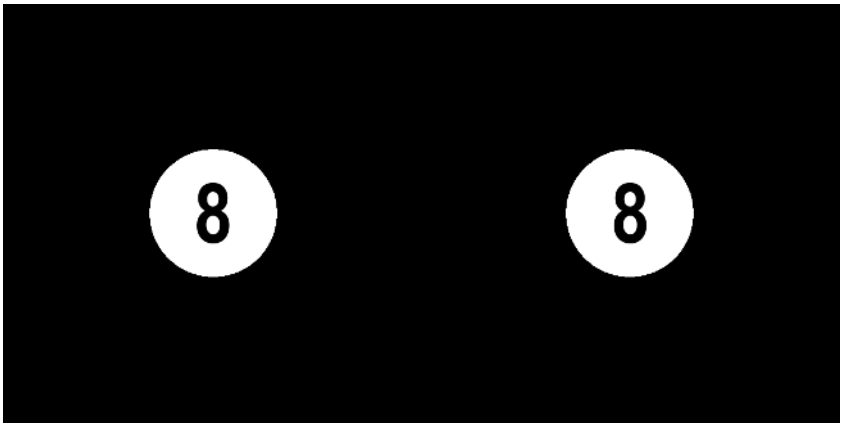
1. **Select the sphere.**
When the sphere is selected, you can see its material as a component in the Inspector window:



We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

2. **Go to our Billiard Ball Textures page on GitHub.**
Go to *https://github.com/head-first-csharp/fifth-edition* and click on the *Billiard Ball Textures* link to browse a folder of texture files for a complete set of billiard balls.

3. **Download the texture for the 8 ball.**
Click on the file *8 Ball Texture.png* to view the texture for an 8 ball. It's an ordinary 1200 × 600 PNG image file that you can open in your favorite image viewer.



We designed this image file so that it looks like an 8 ball when Unity "wraps" it around a sphere.

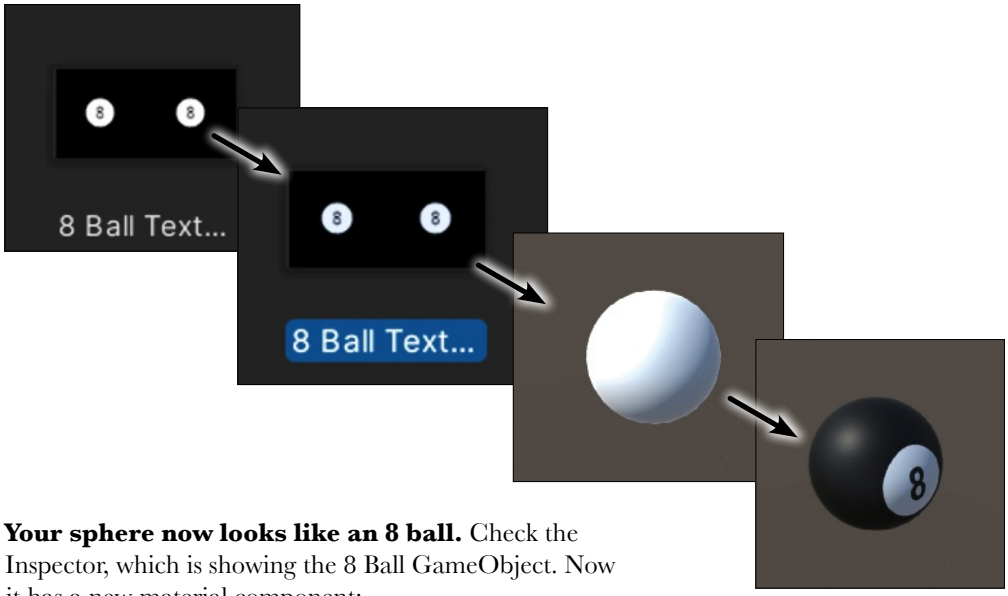
Download the file into a folder on your computer.

(You might need to right-click on the Download button to save the file, or click Download to open it and then save it, depending on your browser.)

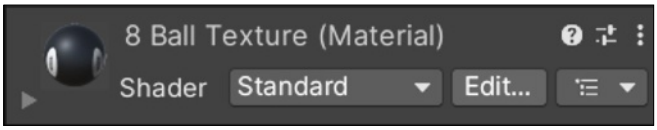
④ **Import the 8 Ball Texture image into your Unity project.**
Right-click on the Assets folder in the Project window, choose **Import New Asset** and import the texture file. You should now see it when you click on the Assets folder in the Project window.

You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

⑤ **Add the texture to your sphere.**
Now you just need to take that texture and "wrap" it around your sphere. Click on 8 Ball Texture in the Project window to select it. Once it's selected, **drag it onto your sphere**.



Your sphere now looks like an 8 ball. Check the Inspector, which is showing the 8 Ball GameObject. Now it has a new material component:



Check your Assets window again. Unity created a new Materials folder in it and added a material called 8 Ball Texture.

I'm learning C# for my job, not to write video games. Why should I care about Unity?

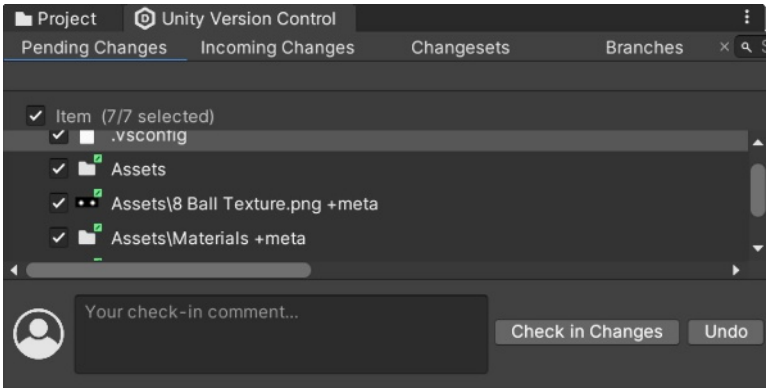
Unity is a great way to really "get" C#.

Programming is a skill, and the more practice you get writing C# code, the better your coding skills will get. That's why we designed the Unity Labs throughout this book to specifically **help you practice your C# skills** and reinforce the C# tools and concepts that you learn in each chapter. As you write more C# code, you'll get better at it, and that's a really effective way to become a great C# developer. Neuroscience tells us that we learn more effectively when we experiment, so we designed these Unity Labs with lots of options for experimentation, and suggestions for how you can get creative and keep going with each lab.

But Unity gives us an even more important opportunity to help get important C# concepts and techniques into your brain. When you're learning a new programming language, it's really helpful to see how that language works with lots of different platforms and technologies. That's why we included both console apps and MAUI apps in the main chapter material, and in some cases even have you build the same project using both technologies. Adding Unity to the mix gives you a third perspective, which can accelerate your understanding of C#.

Do you want to make sure your Unity projects are always backed up? Try Unity Version Control.

Unity Version Control is a version control system that lets you back up your projects to cloud storage that comes free with your Unity account—and it's ***built right into the Unity editor***, which makes it easy for you to use.

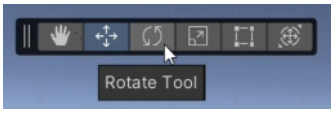


Click the Unity Version Control button in the toolbar to open the Unity Version Control window. The first time you use it, you'll get an option to log in or sign up. When you sign in with your Unity ID, you'll get to a web page where you can sign into your Unity account, then sign up for the free Unity VCS level and join your default organization. Then you can check in changes any time you want.

Go to the Head First C# GitHub page for a free PDF that gives you step-by-step instructions for setting Unity Version Control: *https://github.com/head-first-csharp/fifth-edition*

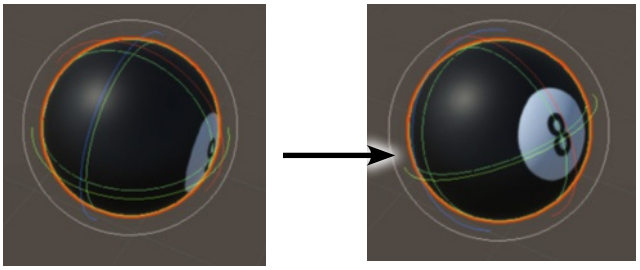
Rotate your sphere

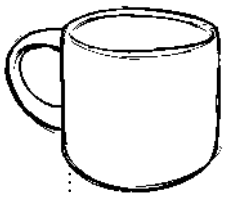
Click the **Rotate tool** in the toolbar. You can use the Q, W, E, R, T, and Y keys to quickly switch between the Transform tools—press E and W to toggle between the Rotate tool and Move Tool.



We switched the Tools overlay to a horizontal view by right-clicking on the two lines and choosing Horizontal. Try it out.

1. **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



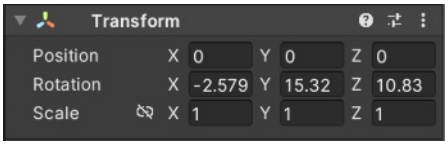


Relax

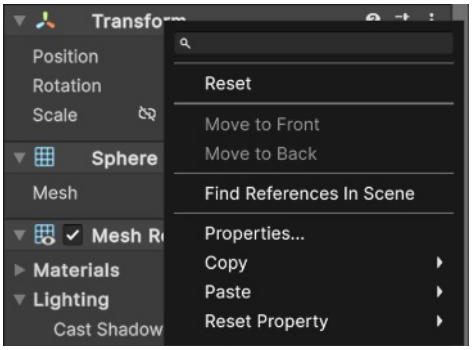
It's easy to reset your windows and scene camera.

If you change your Scene view so you can't see your sphere anymore, or if you drag your windows out of position, just use the Layout dropdown in the upper-right corner to reset the Unity editor to the Wide layout. It will reset the window layout and move the Scene view

2. **Click and drag the green and blue circles to rotate around the Y and Z axes.** The outer white circle rotates the sphere along the axis coming out of the Scene view camera. Watch the Rotation numbers change in the Inspector window.



3. **Open the context menu of the Transform panel in the Inspector window.** Click Reset, just like you did before. It will reset everything in the Transform component back to default values—in this case, it will change your sphere's rotation back to [0, 0, 0].



Click the three dots (or right-click anywhere in the header of the Transform panel) to bring up the context menu. The Reset option at the top of the menu resets the component to its default values.

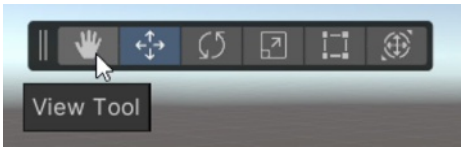
Use these options from farther down in the context menu to reset the position and rotation of a GameObject.

Use File >> Save or Ctrl+S / ⌘S to save the scene right now. Save early, save often!

Move the Scene view camera with the View Tool and Scene Gizmo

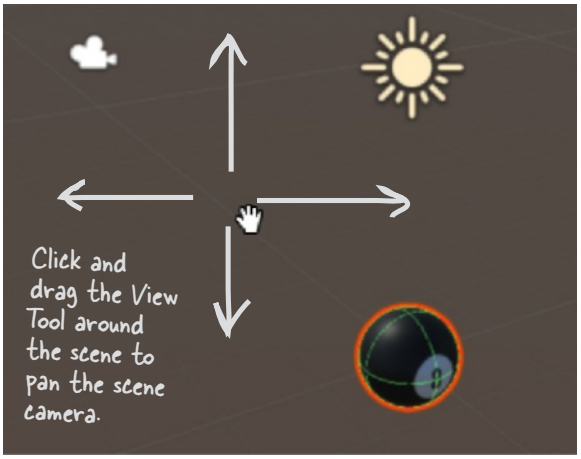
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera in and out.

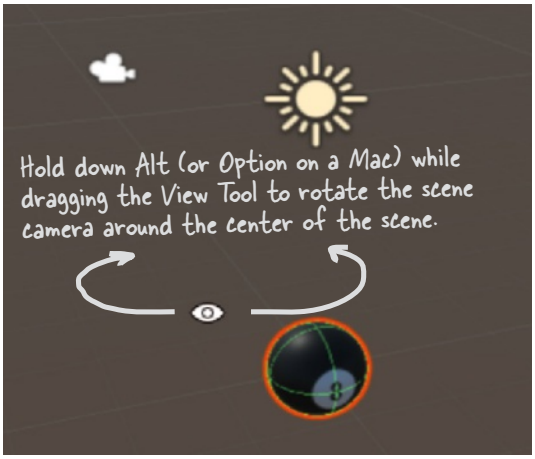
Press Q to select the **View Tool**, or choose it from the toolbar. Your cursor will change to a hand.



Hold down Alt (Windows) or Option (Mac) while dragging and the View Tool turns into an eye and rotates the view around the center of the window.

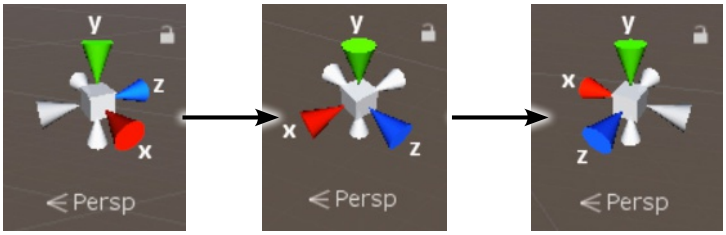
The View Tool pans around the scene by changing the position and rotation of the scene camera. When the View Tool is selected, you can click anywhere in the scene to pan.





When the View Tool is selected, you can ***pan*** the scene camera by **clicking and dragging**, and you can ***rotate*** it by **holding down Alt (or Option) and dragging**. Use the **mouse scroll wheel** to zoom. Holding down the **right mouse button** lets you ***fly through the scene*** using the W-A-S-D keys.

When you rotate the scene camera, keep an eye on the **Scene Gizmo** in the upper-right corner of the Scene window. The Scene Gizmo always displays the camera's orientation—check it out as you use the View Tool to move the Scene view camera. Click on the X, Y, and Z cones to snap the camera to an axis.



Click any of the cones in the Scene Gizmo to snap the camera to an axis. Drag them around to rotate the camera.

The Unity Manual has great tips on navigating scenes: *https://docs.unity3d.com/Manual/SceneViewNavigation.html*

Take a minute and look at this page—it's got some really useful stuff.

there are no Dumb Questions

Q: I'm still not clear on exactly what a component is. What does it do, and how is it different from a GameObject?

A: A GameObject doesn't actually do much on its own. All a GameObject really does is serve as a *container* for components. When you used the GameObject menu to add a sphere to your scene, Unity created a new GameObject and added all of the components that make up a sphere, including a Transform component to give it position, rotation, and scale, a default material to give it its plain white color, and a few other components to give it its shape, and help your game figure out when it bumps into other objects. These components are what make it a sphere.

Q: So does that mean I can just add any component to a GameObject and it gets that behavior?

A: Yes, exactly. When Unity created your scene, it added two GameObjects, one called Main Camera and another called Directional Light. If you click on Main Camera in the Hierarchy window, you'll see that it has three components: a Transform, a Camera, and an Audio Listener. If you think about it, that's all a camera actually needs to do: be somewhere, and pick up visuals and audio. The Directional Light GameObject just has two components: a Transform and a Light, which casts light on other GameObjects in the scene.

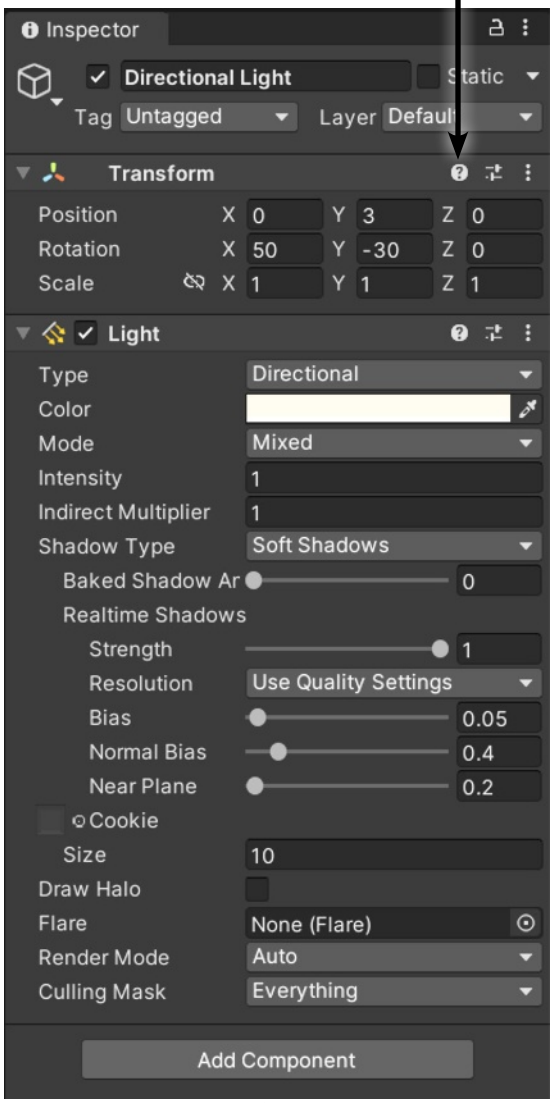
Q: If I add a Light component to any GameObject, does it become a light?

A: Yes! A light is just a GameObject with a Light component. If you click on the Add Component button at the bottom of the Inspector and add a Light component to your ball, it will start emitting light. If you add another GameObject to the scene, it will reflect that light.

Q: It sounds like you're being careful with the way you talk about light. Is there a reason you talk about emitting and reflecting light? Why don't you just say that it glows?

A: Because there's a difference between a GameObject that emits light and one that glows. If you add a Light component to your ball, it will start emitting light—but it won't look any different, because the Light only affects other GameObjects in the scene that reflect its light. If you want your GameObject to glow, you'll need to change its material or use another component that affects how it's rendered.

You can click on the Help icon for any component to bring up the Unity Manual page for it.



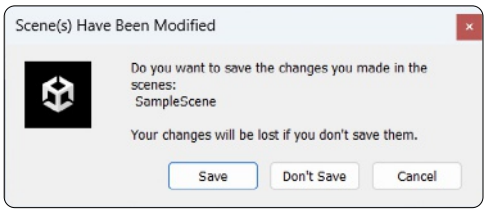
When you click on the Directional Light GameObject in the Hierarchy window, the Inspector shows you its components. It just has two: a Transform component that provides its position and rotation and a Light component that actually casts the light. What do you think you'll use the Add Component button for?

Get creative!

We built these Unity Labs to give you a **platform to experiment on your own with C#** because that's the single most effective way for you to become a great C# developer. This lab ***lays down the foundation*** to start writing Unity code—which you'll do in the next lab. At the end of each Unity Lab, we'll include suggestions for things that you can try on your own. Take some time to experiment with everything you just learned before moving to the next chapter:

- Add a few more spheres to your scene. Try using some of the other billiard ball maps. You can download them all from the same location where you downloaded *8 Ball Texture.png*.
- Try adding other shapes by choosing Cube, Cylinder, or Capsule from the GameObject >> 3D Object menu.
- Experiment with using different images as textures. See what happens to photos of people or scenery when you use them to create textures and add them to different shapes.
- Can you create an interesting 3D scene out of shapes, textures, and lights?

When you're ready to move on to the next chapter, make sure you save your project, because you'll come back to it in the next lab. Unity will prompt you to save when you quit.



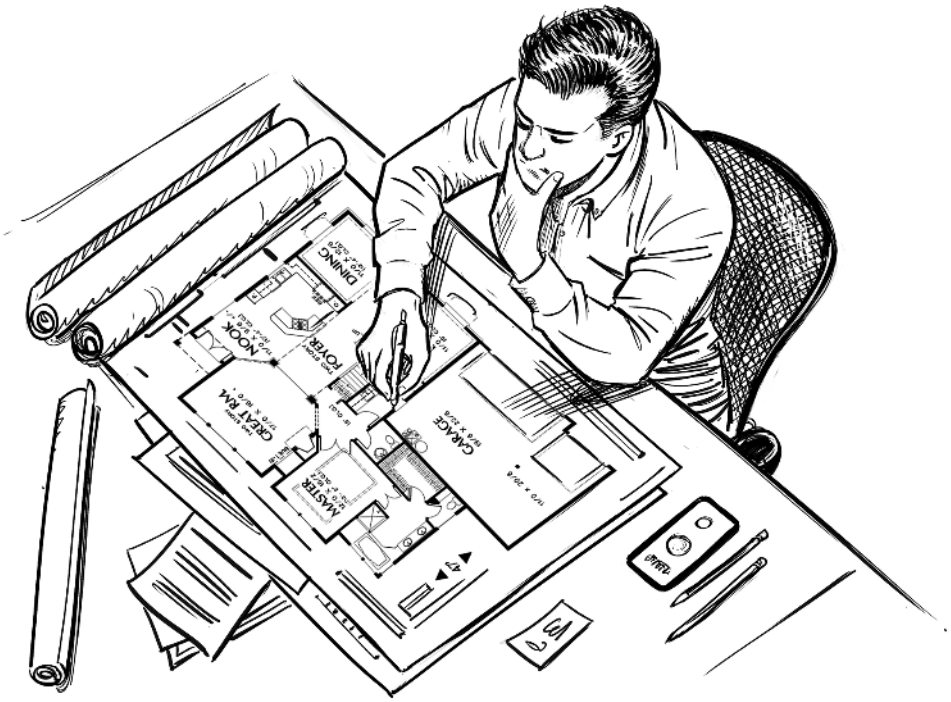
The more C# code you write, the better you'll get at it. That's the most effective way for you to become a great C# developer. We designed these Unity Labs to give you a platform for practice and experimentation.

Bullet Points

- The **Scene view** is your main interactive view of the world that you're creating.
- When you select an object and use the **Move Tool**, Unity displays the **Move Gizmo** that lets you move objects around your scene.
- The **View Tool** lets you pan around the scene. The **Scene Gizmo** always displays the camera's orientation.
- Unity uses **materials** to provide color, patterns, textures, and other visual effects.
- Some materials use **textures**, or image files wrapped around shapes.
- Your game's scenery, characters, props, cameras, and lights are all built from **GameObjects**.
- GameObjects are the fundamental objects in Unity. **Components** are the building blocks for their behavior.
- Every GameObject has a **Transform component** that provides its position, rotation, and scale.
- The **Project window** gives you a folder-based view of your project's assets, including C# scripts and textures.
- The **Hierarchy window** shows all of the GameObjects in the scene.
- **Unity Version Control System (VCS)** is an easy way to back up projects to free cloud storage that comes with a Unity Personal account. Download a PDF to help you set up version control in Visual Studio, VSCode, and Unity for free from our GitHub page: ***https://github.com/head-first-csharp/fifth-edition***

3 namespaces and classes

Organizing your code



Great developers keep their code and data organized.

What's the first thing you do when you're creating an app? You think about what **it's supposed to do**, whether you're solving a problem, creating a game, or just having fun. But it's not always obvious how individual statements fit into your app's bigger picture…and that's where **classes** come in. They let you **organize your code** around the features you're creating and the problems the app needs to solve. Classes can help you **organize your data,** too, by using them to create **objects** that represent any "thing" your app needs to know about—and the classes that you design serve as "blueprints" for the objects used in your app.

Classes help you organize your code

Let's be honest…you're going to write a lot of code throughout this book. And as you keep going through the chapters, your projects will get bigger and bigger. This is a good thing!

Bigger apps present an interesting challenge. The app you built at the end of Chapter 2 had just a few methods. If you create a console app with the same number of methods, there's no reason not to put them all in *Program.cs*.

By the time you get to the end of the book, you'll be creating apps with *dozens* of methods. If you put all of those methods into one big *Program.cs* file, you'll have a hard time remembering which ones do what—and you'll drive yourself crazy trying to sleuth out bugs!

Luckily, C# has an answer for this organizational challenge. Your C# code is organized into **classes**, or units of code that contain methods. You could still put all of your methods into one big class, and many small apps could have just one class. But when you have a lot of code, it makes sense to ***organize your classes based on what they do***. When your classes are organized in a way that's intuitive, it helps you figure out where to add new methods—and it makes sleuthing out bugs a lot easier.

Anatomy of a C# App

Every C# program's code is structured in exactly the same way. All programs use *namespaces*, *classes*, and *methods* to make your code easier to manage.

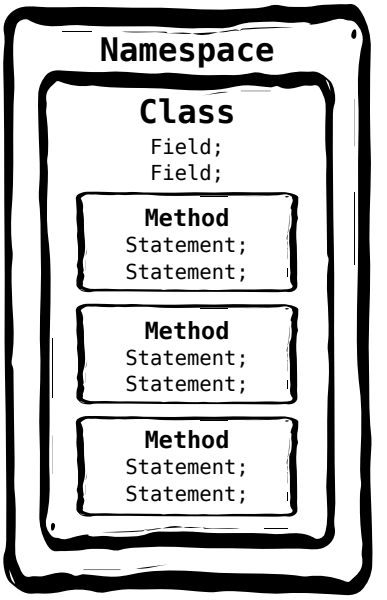
When you create your app, all of the code is inside a *namespace*. This helps keep your classes separate from the ones that come with .NET.

A *class* contains a piece of your program. Some very small programs can have just one class, but most have more.

A class can have *fields*. A field is a variable, except that it's declared outside of the methods so all of the methods in the class can use it.

A class has one or more *methods*. Your methods must live inside a class. The order of the methods in the class file doesn't matter. Method 2 can just as easily come before method 1.

Methods are made up of *statements*—like the ones you used in your apps in the last two chapters.



If code is useful, classes can help you reuse it

Developers have been reusing code since the earliest days of programming, and it's not hard to see why. If you've written a class for one program, and you have another program that needs code that does exactly the same thing, then it makes sense to **reuse** the same class in your new program. So if we were going to build an app called PetManager, we might organize the code using classes called Dog and Cat.

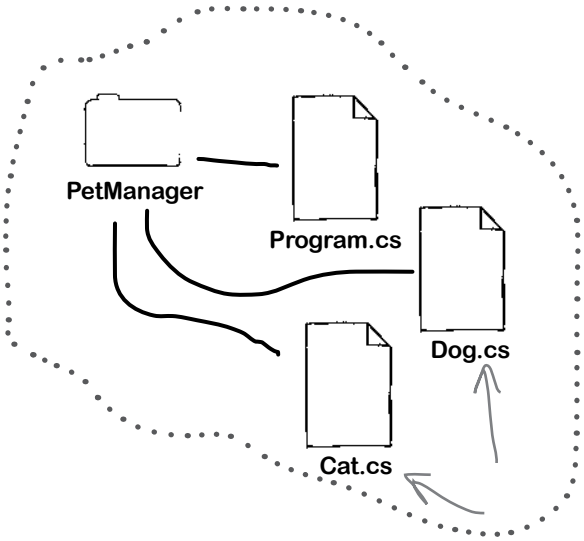
We built the Dog and Cat classes for our PetManagerApp console app...

Dog.cs

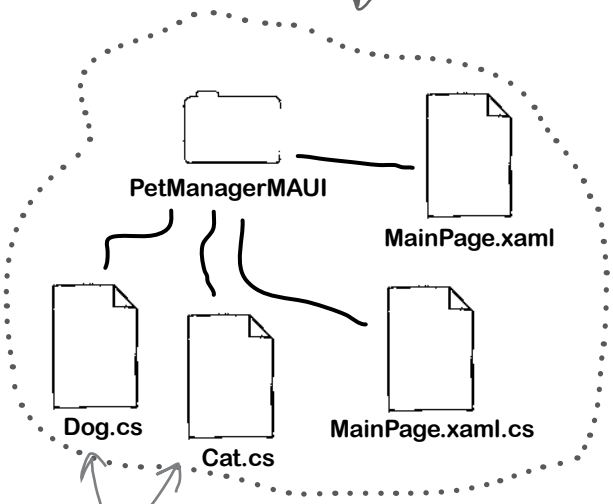
```
namespace Pets {

  class Dog {

    public void Bark() {
       // statements
    }

  }

}
```

Cat.cs

```
namespace Pets {

  class Cat {

    public void Meow() {
       // statements
    }

  }

}
```



...but we discovered we needed exactly the same classes in the MAUI version of our PetManager app, so we reused them.



Since we put our classes in the Pets namespace, we just had to copy the file into the new project and add "using Pets;" to the top of any file with C# code that uses the Dog or Cat classes.

These are the same files with the Dog and Cat classes.

Some methods take parameters and return a value

You've seen methods that do things, like the OperatorExamples method you generated in Chapter 2. Methods can do more than that: they can use **parameters** to get input, do something with that input, and then generate output with a **return value** that can be used by the statement that called the method.

Parameters are values that the method uses as input. They're declared as variables that are included in the method declaration (between the parentheses). The return value is a value that's calculated or generated inside the method, and sent back to the statement that called that method. The type of the return value (like *string* or *int*) is called the **return type**. If a method has a return type, then it must use a **`return` statement**.

Here's an example of a method with two int parameters and an int return type:

```
int Multiply(int factor1, int factor2)
{
    int product = factor1 * factor2;
    return product;
}
```

The return type is int, so the method must return an int value.

This method takes two int parameters called factor1 and factor2 as input. They're treated just like int variables.

The return statement passes the value back to the statement that called the method.

The method takes two **parameters** called `factor1` and `factor2`. It uses the multiplication operator `*` to calculate the result, which it returns using the `return` keyword.

This code calls the Multiply method and stores the result in a variable called `area`:

```
int height = 179;
int width = 83;
int area = Multiply(height, width);
```

You can pass values like 3 and 5 to methods, like this: Multiply(3, 5)—but you can also use variables when you call your methods. It's fine if the variable names don't match the parameter names.

A method's parameters let you give it information that it can use, and its return value lets you use the result of the method in the statement that called it.

Visual Studio helps you explore parameters and return values

In the next app, you'll be using a .NET method called **Console.ReadLine** to get a line that the user types into the console. Once you add the line to your app, you can hover over it to see more about it:

```
Console.ReadLine();
```

This method returns a string? value that holds a line of text that the user typed.

```
string? Console.ReadLine()
Reads the next line of characters from the standard input stream.

Returns:
  The next line of characters from the input stream, or null if no more lines are available.

Exceptions:
  IOException
  OutOfMemoryException
  ArgumentOutOfRangeException
```

This pop-up is called the *Quick Info window*, and it's a really useful part of Visual Studio's IntelliSense system. Read it really carefully. We'll use this window throughout the book as a learning tool. (VSCode has a similar pop-up.)

The IDE will pop up a box telling you what the method does. The very first thing in the box is its return type—in this case, it's a **`string?`** value that holds text. We'll learn a lot more about how strings work in the next chapter. In the meantime, what you need to know is that you call the method like this:

```
string? line = Console.ReadLine();
```

This calls the method to read a line of input, and stores the text the user typed in a variable called **`line`**.

You'll also use a method called int.TryParse(), which you'll use in an **`if`** statement like this:

```
if (int.TryParse(line, out int numberOfCards))
```

```
bool int.TryParse(string? s, out int result) (+ 8 overloads)
Converts the string representation of a number to its 32-bit signed integer
equivalent. A return value indicates whether the conversion succeeded.

Returns:
  true if s was converted successfully; otherwise, false.
```

This method takes two parameters, a string? that contains text to turn into a number, and a result that the value gets saved into.

The IDE is telling you that it takes two parameters, a **`string?`** value and an **`out int`** value. (We'll learn a lot more about the **`out`** keywords later in the book—for now, we'll give you the code to use.)

Do this! → Since you're about to create methods that return values, right now is a perfect time to write some code and use the debugger to *really dig into how the* **`return`** *statement works*:

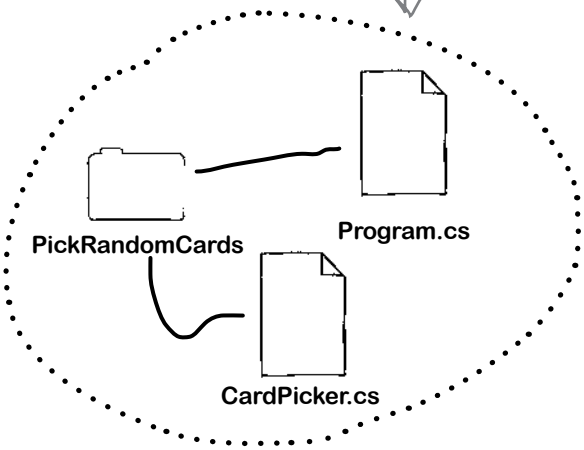
- ★ What happens when a method is done executing all of its statements? See for yourself—open up one of the programs you've written so far, place a breakpoint inside a method, then keep stepping through it.
- ★ When the method runs out of statements, *it* ***returns*** *to the statement that called it* and continues executing the next statement after that.
- ★ A method can also include a **`return`** statement, which causes it to immediately exit without executing any of its other statements. Try adding a **`return`** statement in the middle of a method, then stepping over it.

Let's build an app that picks random cards

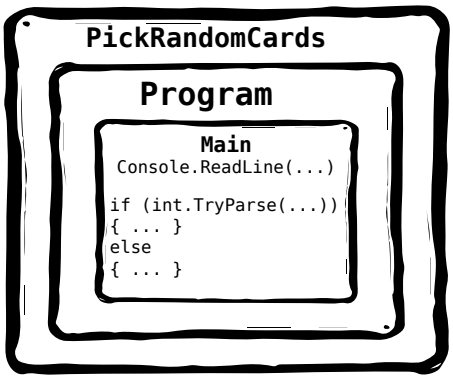
In the first project in this chapter, you're going to build a console app called PickRandomCards that lets you pick random playing cards.

Let's use it as a way to **start using classes**. Here's what its structure will look like:

Your PickRandomCards app will have a second C# class file called CardPicker.cs, which contains the code for the CardPicker class.

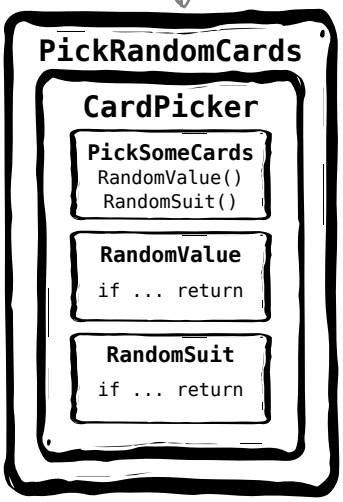


You're going to create a console app that *has a Main method* instead of top-level statements (which we'll talk more about). Your *Program.cs* file will contain a class. That class will have a method called Main, which is the first thing that gets run when you start the app.



When you create your app you'll call it PickRandomCards, so Visual Studio will create a namespace for you that matches the name of the app.

Your Main method will have all the code that communicates with the user, displaying text and getting input. The code that has to do with picking random cards will be in a *class called CardPicker*.



The CardPicker class doesn't have any fields, and that's OK! We'll talk more about fields later in the chapter.

You'll use an array to store the cards

Your PickSomeCards method will use string values to represent playing cards. Let's say you want to use your CardPicker class to pick five random cards and store them in a variable called `cards`. Here's how you would do that:

```
string[] cards = CardPicker.PickSomeCards(5);
```

There's a lot going on in that line of code, so let's break it down. The PickSomeCards method is in the CardPicker class, but we're calling it from a top-level statement, so we need to use the class name to call it:

```
string[] cards = CardPicker.PickSomeCards(5);
```

We just learned about how methods can take parameters. The PickSomeCards method takes an int parameter, and we're passing the method the value 5 to tell it to pick five cards:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The first part of the statement declares the cards variable. We just learned about return values—so the method will return a value that gets stored in the `cards` variable. But something looks different about it:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The `cards` variable has a type that you haven't seen yet. Look closely at the type:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The square brackets `[]` mean that it's an **array of strings**. Arrays let you use a single variable to store multiple values—in this case, strings with playing cards—which will get stored in the `cards` variable.

You can use a **collection expression** to create an array of values by putting comma-separated values between two square brackets `[]`. Here's an example of a string array that the PickSomeCards method might return:

```
[
    "10 of Diamonds",
    "6 of Clubs",
    "7 of Spades",
    "Ace of Diamonds",
    "Ace of Hearts",
]
```

This array of strings has five values in it. Each value is a separate string that has the name of a card.

After your array is generated, you can use a `foreach` loop to write each of the cards to the console:

```
foreach (string card in cards) {
    Console.WriteLine(card);
}
```

For the previous array, running that `foreach` loop will generate this output:

```
10 of Diamonds
6 of Clubs
7 of Spades
Ace of Diamonds
Ace of Hearts
```

Create an app with a Main method

When you created console apps in the first two chapters, the IDE generated a two-line *Program.cs* file:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

You may not have realized it at the time, but you were taking advantage of a very useful feature of C# called **top-level statements** that lets you start creating a console app with a simple file that contains a set of statements that get executed in order.

Top-level statements are really convenient! A good way to understand what they do for you and how they work is to ***create a C# app that doesn't use top-level statements***.

Do this!

Create your PickRandomCards app without top-level statements

Use Visual Studio to **create a new console app called PickRandomCards**. But unlike previous chapters, when you're going through the steps to create the app, keep an eye out for a checkbox like this and make sure that it's checked:

☑ Do not use top-level statements ⓘ

Make sure you check the "Do not use top-level statements box" when you create your app; otherwise, your Program.cs file won't have a Program class with Main method.

When you create your new app, your *Program.cs* file should look like this:

```
namespace PickRandomCards
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

The very first statement that gets executed in an app is called its *entry point*. The entry point in an app with top-level statements is the first statement in *Program.cs*. In an app without top-level statements, the entry point is the Main method.

When the IDE created your app and generated the *Program.cs* file, **it added a class called Program**. This program was generated with one method called Main. The code inside the Main method is the familiar statement that prints "Hello, World!" to the console.

Run your app—it should look very familiar. Your new app does exactly the same thing as the "Hello, World!" app you created in Chapter 1. But instead of starting at the first statement in the *Program.cs* file, the first thing your app does is execute the Main method.

Your app can only have one entry point. If you add another class with a Main method, your code won't build.

Watch it!

Visual Studio remembers your checkbox choices.

The next time you create a Console App project in Visual Studio, it may remember that you checked the "Do not use top-level statements" box and check it again for you. Make sure it's unchecked the next time you create a console app.

Top-Level Statements Behind the Scenes

Here's what happens when you use *top-level statements*

Here's the very first app that you created in Chapter 1:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

The first line is a comment, so there's actually only one statement in this app. When you build the app, the *compiler*—the part of Visual Studio that turns your C# code into something that your operating system can execute—reads all of the lines in the top-level statement and adds them to a class. It creates a class that looks like the one that you just saw in your PickRandomCards app:

```
internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

The top-level statements in this app are in the *global namespace*, a default namespace that contains any classes that aren't explicitly put into a namespace. You can put any class into the global namespace by leaving the namespace declaration off.

That looks a lot like the Program class that Visual Studio just created in your PickRandomCards app. But there's one difference—can you spot it?

Here's the Program class from your PickRandomCards app without top-level statements—we've made text that's the same a lighter color so you can see the difference:

```
namespace PickRandomCards
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

If you're using VSCode and don't have the option to create a console app that does not use top-level statements, replace the contents of Program.cs with this code.

In an app with top-level statements, the code in your *Program.cs* file *is not in a namespace*. Every method in a C# program must be inside a class, but it's OK for classes to be outside of namespaces.

We'll learn more about how to work with namespaces later in this chapter.

The rest of the console apps in this book will use top-level statements, so when you create the next one make sure to uncheck the "Do not use top-level statements" checkbox.

At the time we're writing this, VSCode* does not have the option *to create a console app that does not use top-level statements. Just create your console app just like you did in Chapters 1 and 2, then replace the contents of Program.cs with the code we just showed you.

Add a class called CardPicker to your app

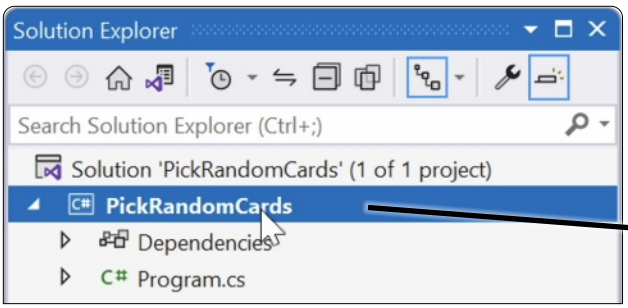
The next thing you'll do is add a class called CardPicker to your app. Here's a **class diagram** that shows information about the class you'll build, including the methods that you'll add to your CardPicker class:

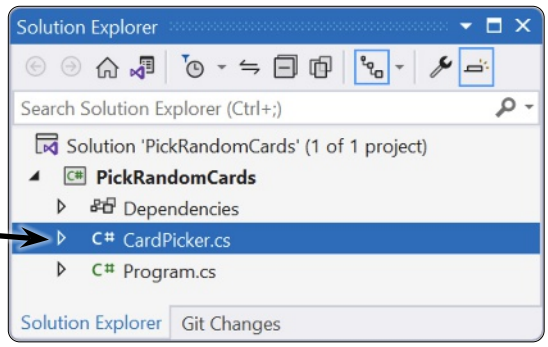
This is a *class diagram*. It's a rectangle with the class name on top and a list of its methods on the bottom. Your CardPicker class will have three methods named PickSomeCards, RandomSuit, and RandomValue.

Do this!

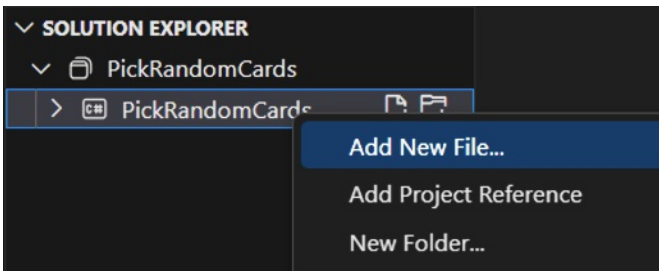
Luckily, both Visual Studio and VSCode will help you add a new class to your project.

For Visual Studio: Right-click on the **PickRandomCards project** in the Solution Explorer—it's the ***second*** row, underneath the solution, which has the same name. When the right-mouse menu pops up, **choose Add >> Class** (or choose *Add Class...* from the Project menu). (If you don't want to do all that clicking, you can also choose Add Class from the Project menu instead of right-clicking on the project name to jump straight to the filename prompt.) **Enter `CardPicker.cs`** when you're asked for a filename. You should now see a new file called *CardPicker.cs* in your Solution Explorer.





For VSCode: Right-click on the **PickRandomCards project** in the Solution Explorer—it's the ***second*** row, underneath the solution, which has the same name. When the right-mouse menu pops up, **choose Add New File**, VSCode will display a list of templates at the top of the window and ask you to choose one. Choose **Class**. VSCode will then display a prompt at the top of the window asking you for a filename. **Enter `CardPicker.cs`**. You should now see a new file called *CardPicker.cs* in your Solution Explorer.

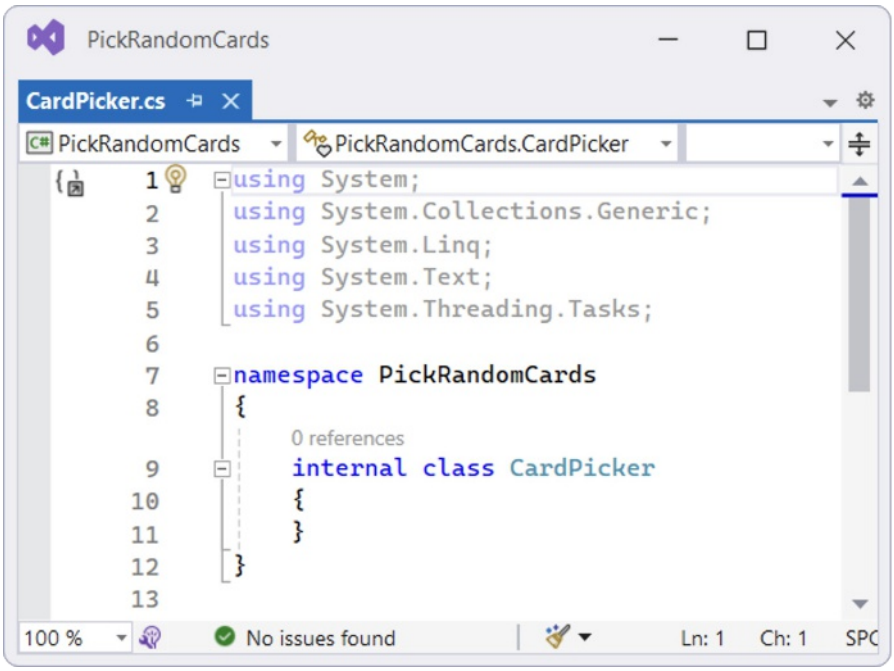


When you right-click on the project name and choose "Add New File..." or click the add file icon next to the project name, VSCode will prompt you for the type of file to add—choose Class. Make sure you're in the *Solution Explorer*, not the Explorer.

Sharpen your pencil

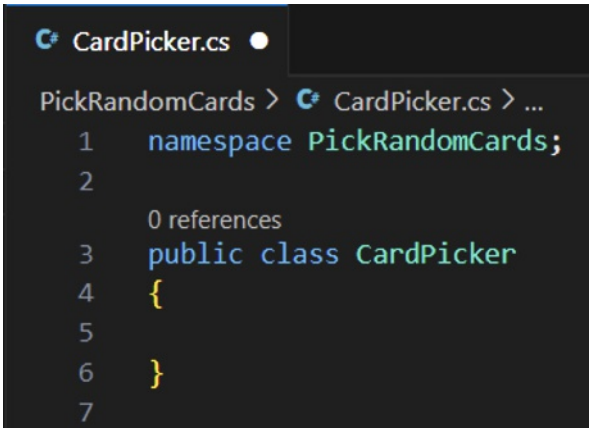
Double-click on CardPicker.cs in the Solution Explorer to open it. The code in the file will look slightly different depending on whether you used Visual Studio or VSCode.

Here's what your file will look like if you're using Visual Studio:



You may not see the "using" lines 1 through 5 in Visual Studio. An update to .NET may have removed them since we took that screenshot. We kept them here for this "Sharpen" exercise.

Here's what it will look like if you're using VSCode:



Compare the code in these two screenshots. You should be able to spot three important differences. Write them down:

.....

.....

.....

.....

.....

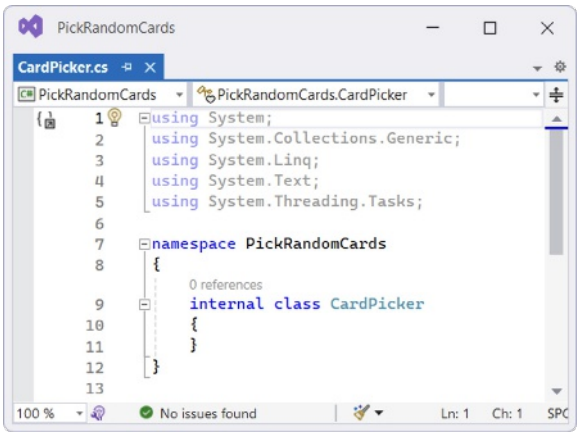
.....

.....

.....

Sharpen your pencil Solution

We asked you to compare the code in two screenshots. Here's the screenshot from Visual Studio:



Did you spot these three differences?

Difference #1: The first five lines in the Visual Studio CardPicker.cs file that start with 'using' are not in the VSCode file.

Difference #2: The 'namespace' line in the VSCode file ends with a semicolon while the same line in Visual Studio is followed by { brackets }.

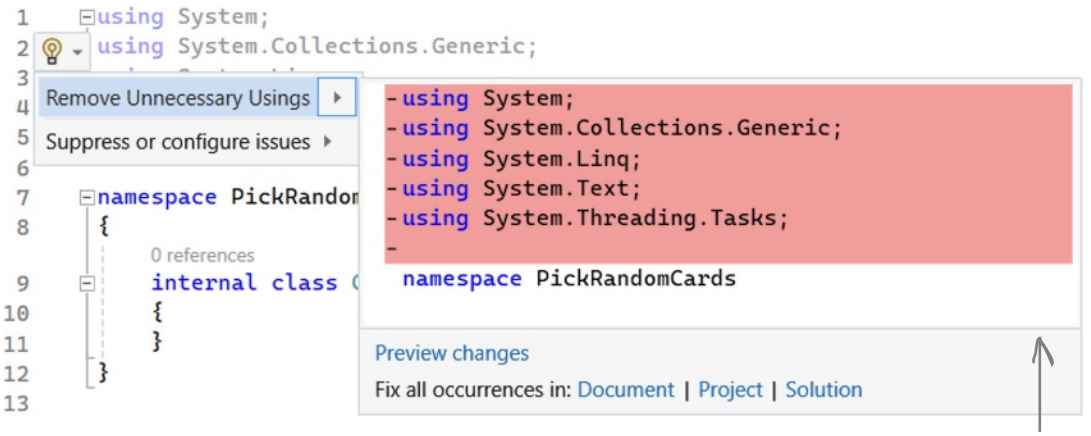
Difference #3: The 'class CardPicker' line starts with 'internal' in VS and 'public' in VSCode.

Use Quick Actions to remove unnecessary using lines

If you're using Visual Studio, you'll see five lines at the top of your *CardPicker.cs* file that start with `using`. These are **using directives**, and you'll learn about them later in the book. But for now, they're not needed.

Luckily, Visual Studio makes it easy to remove those unnecessary using directives from your file, which is good for us because we want to keep things simple and easy to read. Click anywhere in the top five lines. Visual Studio will display the Quick Actions icon to indicate that there's a quick action available.

Choose the **Remove Unnecessary Usings quick action**. Visual Studio will remove the five `using` lines.



If you don't have "using" lines, you can skip this step.

Convert between namespace styles

Let's compare the Visual Studio and VSCode versions of the *CardPicker.cs* file again, this time with the unnecessary using directives at the top of the Visual Studio version removed.

If you're using Visual Studio, your *CardPicker.cs* file uses a **block-scoped namespace**. The `namespace` keyword is followed by the name of the namespace (`PickRandomCards`), which is followed by a block of code inside `{` brackets `}` and everything between those brackets is in the `PickRandomCards` namespace.

```
namespace PickRandomCards
{
    internal class CardPicker
    {
    }
}
```

If you're using VSCode, your *CardPicker.cs* file uses a **file-scoped namespace**. The `namespace` keyword is followed by the name of the namespace (`PickRandomCards`) and a semicolon. Everything in the entire file is in the `PickRandomCards` namespace.

```
namespace PickRandomCards;

public class CardPicker
{

}
```

This is the second difference you spotted in the "Sharpen your pencil" exercise. Both ways to define a namespace are valid. They're two different ways to say the same thing: the CardPicker class is inside the PickRandomCards namespace.

Click on the `namespace` line in your file. You'll see a light bulb or screwdriver icon indicating that there's a quick action available (the icon looks a little different in VSCode). Click the icon (or press Ctrl+. or ⌘.) to bring up the Quick Actions menu:

- If your file has a block-scoped namespace, you'll see a **Convert to file-scoped namespace** quick action.
- If your file has a file-scoped namespace, you'll see a **Convert to block-scoped namespace** quick action.

Choose the quick action to convert to the other namespace style. Then do the same thing and convert it back. Choose the style you like the best—they both work exactly the same way. You can switch namespace styles at any time.

Relax

We'll usually use file-scoped namespaces in the code in this book.

Your code will behave exactly the same whether you use block-scoped or file-scoped namespaces. We'll usually use file-scoped namespaces when we're showing the code in this book, especially in Console App projects, because it takes up less room on the page: file-scoped namespaces don't need the extra lines for the opening and closing brackets, and the code doesn't need to be indented as much.

Also—we didn't talk about the third difference from the "Sharpen your pencil" exercise. The code generated by Visual Studio has the **`internal`** keyword, while the code generated by VSCode has the **`public`** keyword. Those are called **access modifiers**. We'll talk more about them later in the book.

Use the new keyword to create an array of strings

Do this!

Let's say you want to create an array of five strings and store it in a variable called `myStrings`. You can use the **`new` keyword** to create a new array of strings. You can create your array of five strings like this:

```
string myStrings = new string[5];
```

You can also use a variable, field, or method parameter instead of a number. Your PickSomeCards method has a parameter called `numberOfCards`—you'll use that parameter in your new statement: `new string[numberOfCards];`

The PickSomeCards method will pick five random cards. Each of the cards will have a random value and a random suit, so the class will also have two more methods that generate the value and suit for each card.

① Add a new PickSomeCards method to your CardPicker class.

Put your cursor between the curly brackets, press Enter to add a space, and *carefully* **type in this method**:

```
internal class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {

    }
}
```

If you're using VSCode you'll see 'public' instead of 'internal' here. That's OK!

Make sure you include the "public" and "static" keywords. We'll talk more about them later in the chapter.

If you carefully entered your method declaration exactly as it appears here, you should see a red squiggly underline underneath PickSomeCards. What do you think it means?

② Create a new array of strings and store it in a variable called `pickedCards`.

We saw earlier that the PickSomeCards method will return an array of strings, so the first thing we'll need is an array of strings to return. Add this line of code to your method:

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
}
```

You'll see a red squiggly line under PickSomeCards. Visual Studio is telling you that your method is supposed to return something, but there's no corresponding return statement.

Now the method has a string array to work with.

③ Add a `foreach` loop to set the value of each card in the array.

Your method has an array of strings. Now it needs to set them. Add this `foreach` loop—it will call two methods called RandomValue and RandomSuit. Those methods don't exist yet, but that's OK.

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
    for (int i = 0; i < numberOfCards; i++)
    {
        pickedCards[i] = RandomValue() + " of " + RandomSuit();
    }
}
```

The RandomValue and RandomSuit methods don't exist yet, so Visual Studio will warn you about them too.

4 Finish the method by adding a `return` statement.

Add a `return` statement to send the pickedCards array back to the statement that called the method.

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
    for (int i = 0; i < numberOfCards; i++)
    {
        pickedCards[i] = RandomValue() + " of " + RandomSuit();
    }
    return pickedCards;
}
```

Adding the return statement makes the warning on the method declaration line go away, but the warnings for the two method calls are still there.

5 Generate the RandomValue and RandomSuit methods.

In Chapter 2, you generated a method called OperatorExamples. Follow exactly the same steps to **generate a method in the CardPicker class called RandomSuit**. Then do exactly the same thing to generate a method called RandomValue.

A method can have *more than one* `return` *statement*, and when it executes one of those statements it immediately returns, and does not execute any more statements in the method.

6 Implement the RandomSuit method.

Every card has a suit: hearts, clubs, spades, or diamonds. The RandomSuit method will pick a suit at random, store it in a string, and return it. It will use the same random number generator, Random.Shared, that you used in Chapter 1 to pick emoji from a list. The random number generator's Next method can take two parameters: `random.Next(1, 5)` returns a number that's at least 1 but less than 5—in other words, calling `Random.Shared.Next(1, 5)` returns a random number from 1 to 4.

Let's add code to your RandomSuit method that takes advantage of `return` statements to stop executing the method as soon as it finds a match—we added a comment to each line to explain how it works:

```
private static string RandomSuit()
{
    int value = Random.Shared.Next(1, 5);  // pick a random number from 1 to 4
    if (value == 1) return "Spades";       // if it's 1 return the string Spades
    if (value == 1) return "Hearts";       // if it's 2 return the string Hearts
    if (value == 3) return "Clubs";        // if it's 3 return the string Clubs
    return "Diamonds";                     // otherwise return the string
}
```

You used a Random.Shared statement in Chapter 1 to choose random emoji from a list.

The return statement causes your method to stop immediately and go back to the statement that called it.

7 **Implement the RandomValue method.**

Every playing card can have one of 13 values—ace, 2 through 10, jack, queen, or king. Here's the RandomValue method that generates a random value. Look closely at it. Can you figure out how it works?

```
private static string RandomValue()
 {
     int value = Random.Shared.Next(1, 14);
     if (value == 1) return "Ace";
     if (value == 11) return "Jack";
     if (value == 12) return "Queen";
     if (value == 13) return "King";
     return value.ToString();
 }
```

Notice how your method returns `value.ToString()` and not just `value`? That's because `value` is an int variable, but the RandomValue method was declared with a string return type, so we need to convert `value` to a string. You can add `.ToString()` to any variable or value to convert it to a string.

Here's the code for your finished CardPicker class

Your CardPicker class is in the PickRandomCards namespace and has the methods that we just added:

```
namespace PickRandomCards;

internal class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }

    private static string RandomSuit()
    {
        int value = Random.Shared.Next(1, 5);  // pick a random number from 1 to 4
        if (value == 1) return "Spades";       // if it's 1 return the string Spades
        if (value == 2) return "Hearts";       // if it's 2 return the string Hearts
        if (value == 3) return "Clubs";        // if it's 3 return the string Clubs
        return "Diamonds";                     // otherwise return the string
    }

    private static string RandomValue()
    {
        int value = Random.Shared.Next(1, 14);
        if (value == 1) return "Ace";
        if (value == 11) return "Jack";
        if (value == 12) return "Queen";
        if (value == 13) return "King";
        return value.ToString();
    }
}
```

We're using a file-scoped namespace to make the code take up less space on the page. We also stuck with the "internal" access modifier from Visual Studio, but if you're using VSCode and have "public" that's fine too.

Visual Studio will *automatically indent* your code for you so it looks like ours. If you're using VSCode, press Alt+Shift+F or ⌥+Shift+F to automatically fix the indentation in your file.

We added these comments to help you understand how the RandomSuit method works. Try adding similar comments to the RandomValue method to explain how it works. We put the comment on each line and used spaces to line them up. You can also put them on separate lines if you think it looks better or is easier to read.

It's OK if your methods are in a different order.

Exercise

Now that your CardPicker class has a method to pick random cards, you've got everything you need to finish your console app by **filling in the Main method**. You just need a few useful methods to make your console app read a line of input from the user and use it to pick a number of cards.

Useful method #1: Console.Write

You've already seen the Console.WriteLine method. Here's its cousin, Console.Write, which writes text to the console but doesn't add a new line at the end. You'll use it to display a message to the user:

```
Console.Write("Enter the number of cards to pick: ");
```

Useful method #2: Console.ReadLine

The Console.ReadLine method reads a line of text from the input and returns a string. You'll use it to let the user tell you how many cards to pick:

```
string? line = Console.ReadLine();
```

We showed you this line of code earlier in the chapter.

Useful method #3: int.TryParse

Your CardPicker.PickSomeCards method takes an int parameter. The line of input you get from the user is a string, so you'll need a way to convert it to an int. You'll use the int.TryParse method for that:

```
if (int.TryParse(line, out int numberOfCards))
{
  // this block is executed if line COULD be converted to an int
  // value that's stored in a new variable called numberOfCards
}
else
{
   // this block is executed if line COULD NOT be converted to an int
}
```

We also showed you this line of code earlier in the chapter.

Put it all together

Your job is to take these three new pieces and put them together in a brand-new Main method for your console app. Modify your *Program.cs* file and replace the "Hello, World!" line in the Main method with code that does this:

- Use Console.Write to ask the user for the number of cards to pick.
- Use Console.ReadLine to read a line of input into a string variable called `line`.
- Use int.TryParse to try to convert it to an int variable called `numberOfCards`.
- If the user input ***could be converted*** to an int value, use your CardPicker class to pick the number of cards that the user specified: `CardPicker.PickSomeCards(numberOfCards)`. Use a string[] variable to save the results, then use a `foreach` loop to call Console.WriteLine on each card in the array. Flip back to Chapter 1 to see an example of a `foreach` loop—you'll use it to loop through every element of the array. Here's the first line of the loop: `foreach (string card in CardPicker.PickSomeCards(numberOfCards))`
- If the user input ***could not be converted***, use Console.WriteLine to write a message to the user indicating that the number was not valid.

Exercise Solution

Here's the Main method for your console app. It replaces the one that Visual Studio created for you that prints "Hello, World!" This method prompts the user for the number of cards to pick, attempts to convert it to an int, and then uses the PickSomeCards method in the CardPicker class to pick that number of cards. PickSomeCards returns each of the picked cards in an array of strings, so it uses a `foreach` loop to write each of them to the console.

```
static void Main(string[] args)
{
    Console.Write("Enter the number of cards to pick: ");
    string? line = Console.ReadLine();
    if (int.TryParse(line, out int numberOfCards))
    {
        string[] cards = CardPicker.PickSomeCards(numberOfCards);
        foreach (string card in cards)
        {
            Console.WriteLine(card);
        }
    }
    else
    {
        Console.WriteLine("Please enter a valid number.");
    }
}
```

We gave you these lines of code.

This is just like the code we showed you earlier, except instead of passing a number like 5 to the method, you're passing it the numberOfCards variable.

Here's what it looks like when you run your console app:

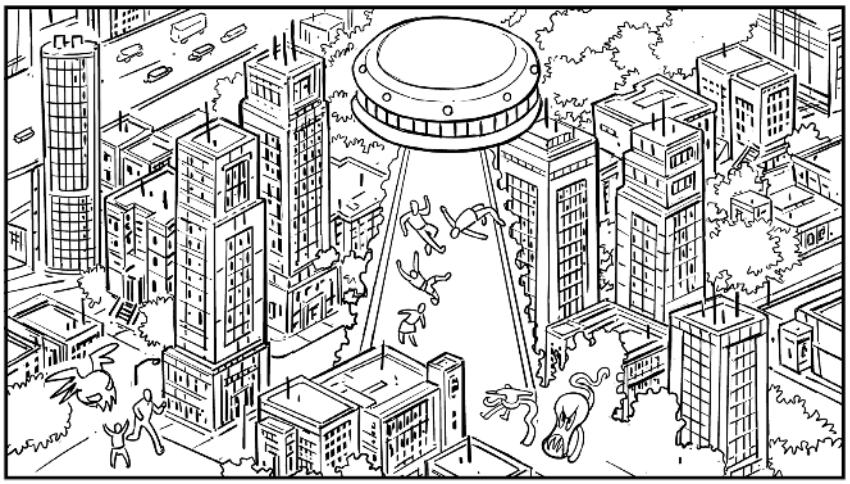
```
Microsoft Visual Studio Debu
Enter the number of cards to pick: 13
Queen of Hearts
8 of Clubs
6 of Diamonds
King of Spades
5 of Diamonds
8 of Diamonds
9 of Clubs
8 of Hearts
5 of Spades
King of Clubs
2 of Clubs
4 of Spades
9 of Spades

C:\Users\andrewstellman\source\repos\PickRandomCards\PickRandomCards\bin\Debug\net6.0\
PickRandomCards.exe (process 4940) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugg
ing->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

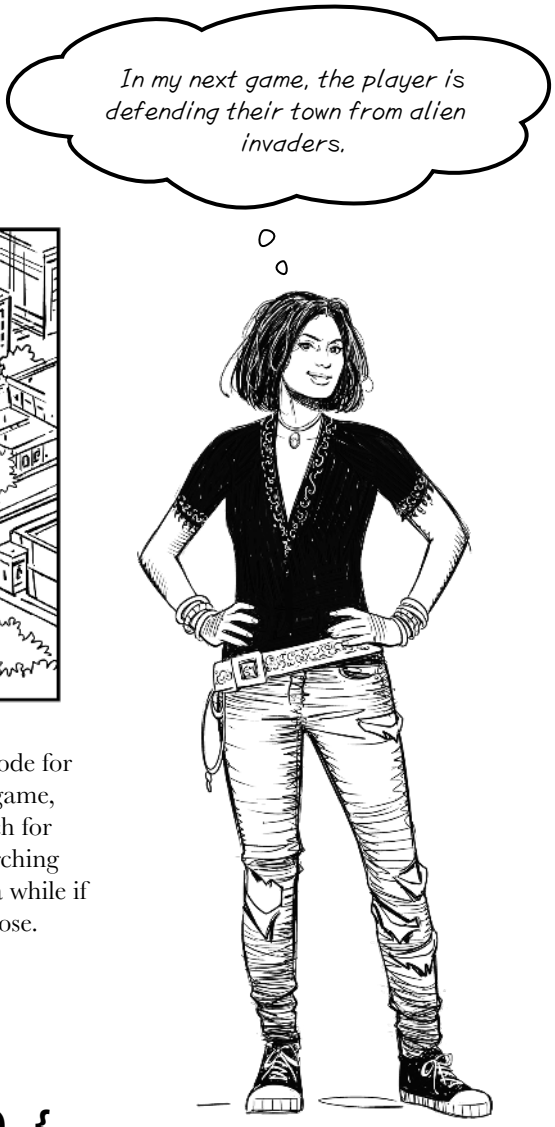
If you're using VSCode, you'll interact with your console app in the Terminal at the bottom of the window. *Click inside the Terminal panel* and type the number of random cards you want your app to pick. If your app is running in the Debug Console window, go back to Chapter 1 and follow the instructions to change the C# debug console setting.

Take the time to really understand how this program works—this is a great opportunity to use the Visual Studio or VSCode debugger to help you explore your code. Place a breakpoint on the first line of the Main method, then use Step Into to step through the entire program. Add a watch for the `value` variable, and keep your eye on it as you step through the RandomSuit and RandomValue methods.

Ana's working on her next game

Meet Ana. She's an indie game developer. Her last game sold thousands of copies, and now she's getting started on her next one.

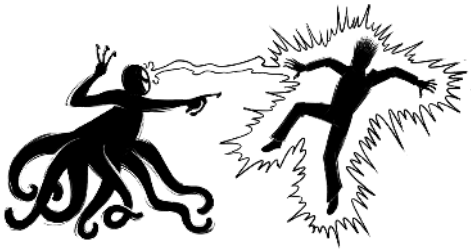


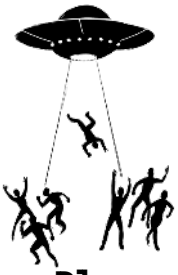


Ana's started working on some **prototypes**. She's been working on the code for the alien enemies that the player has to avoid in one exciting part of the game, where the player needs to escape from their hideout while the aliens search for them. Ana's written several methods that define the enemy behavior: searching the last location where the player was spotted, giving up the search after a while if the player wasn't found, and capturing the player if the enemy gets too close.

```
SearchForPlayer();
```

```
if (SpottedPlayer()) {
    CommunicatePlayerLocation();
}
```

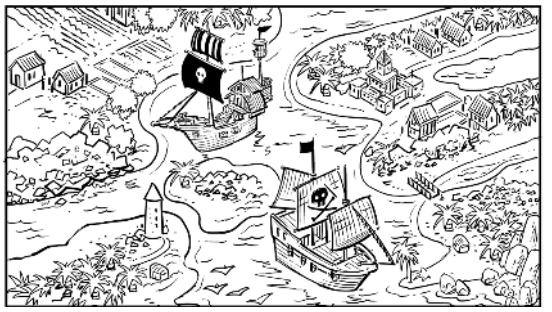


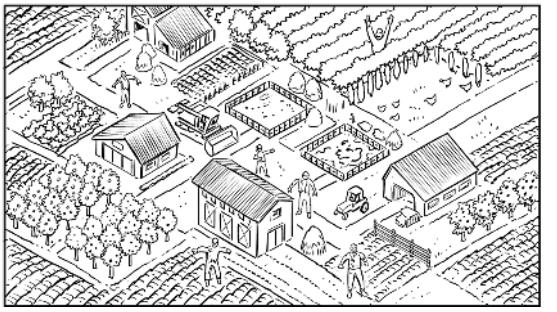


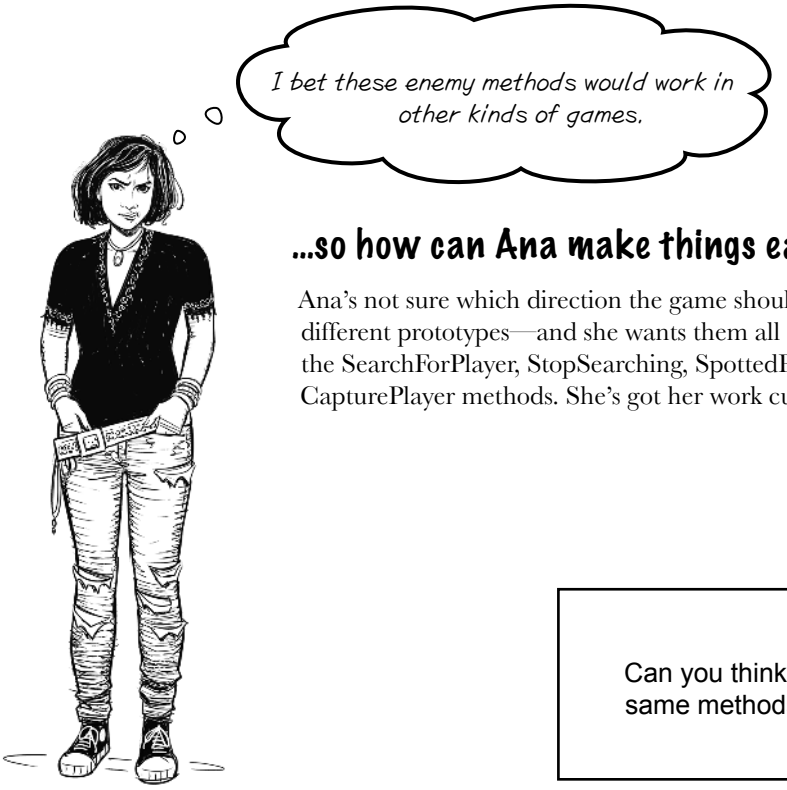
```
CapturePlayer();
```

Ana's game is evolving...

The humans versus aliens idea is pretty good, but Ana's not 100% sure that's the direction she wants to go in. She's also thinking about a nautical game where the player has to evade pirates. Or maybe it's a zombie survival game set on a creepy farm. In all three of those ideas, she thinks the enemies will have different graphics, but their behavior can be driven by the same methods.







...so how can Ana make things easier for herself?

Ana's not sure which direction the game should go in, so she wants to make a few different prototypes—and she wants them all to have the same code for the enemies, with the SearchForPlayer, StopSearching, SpottedPlayer, CommunicatePlayerLocation, and CapturePlayer methods. She's got her work cut out for her.

Brain Power

Can you think of a good way for Ana to use the same methods for enemies in different prototypes?

I put all of the enemy behavior methods into a single Enemy class. Can I **reuse the class** in each of my three different game prototypes?

Enemy
SearchForPlayer SpottedPlayer CommunicatePlayerLocation StopSearching CapturePlayer

Game Design...and Beyond

Prototypes

A **prototype** is an early version of your game that you can play, test, learn from, and improve. A prototype can be a really valuable tool to help you make changes early. Prototypes are especially useful because they let you ***r*apidly experiment with a lot of different ideas** before you've made permanent decisions.

- The first prototype is often a **paper prototype**, where you lay out the core elements of the game on paper. For example, you can learn a lot about your game by using sticky notes or index cards for the different elements of the game, and drawing out levels or play areas on large pieces of paper to move them around.
- One good thing about building prototypes is that they help you get from an idea to a working, playable game very quickly. You ***learn the most about a game*** (or any kind of program) when you get working software into the hands of your players (or users).
- Most games will go through **many prototypes**. This is your chance to try out lots of different things and learn from them. If something doesn't go well, think of it as an experiment, not a mistake.
- **Prototyping is a skill**, and just like any other skill, you get better at it with practice. Luckily, building prototypes is also fun, and a great way to get better at writing C# code.

Prototypes aren't just used for games! When you need to build **any kind of app**, it's often a great idea to build a prototype first to experiment with different ideas.

Build a paper prototype for a classic game

Paper prototypes are really useful for helping you figure out how a game will work before you start building it, which can save you a lot of time. There's a fast way to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made...but you can choose any game you'd like! Here's what to do next.

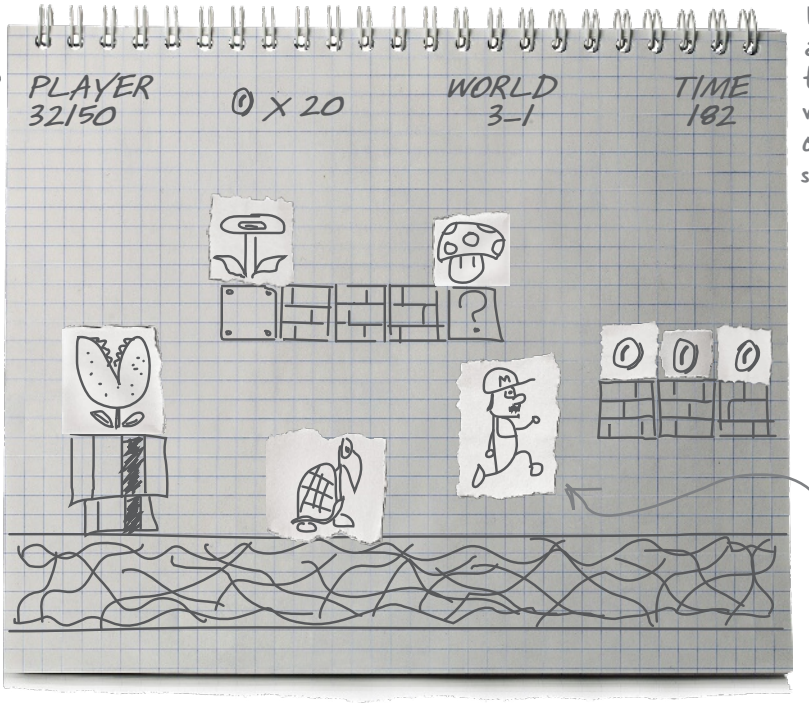
Draw this!

1. **Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.

2. **Tear small scraps of paper and draw the moving parts.** In our prototype, we drew the characters, the piranha plant, the mushroom, the fire flower, and the coins on separate scraps. If you're not an artist, that's absolutely fine! Just draw stick figures and rough shapes. Nobody else ever has to see this!

3. **"Play" the game.** This is the fun part! Try to simulate player movement. Drag the player around the page. Make the nonplayer characters move too. It helps to spend a few minutes playing the game, then go back to your prototype and see if you can reproduce the motion as closely as possible. (It will feel a little weird at first, but that's OK!)

The text at the top of the screen is called the HUD, or head-up display. It's usually drawn on the background in a paper prototype.



When the player catches a mushroom he grows to double his size, so we also drew a small character on a separate scrap of paper.

The ground, bricks, and pipe don't move, so we drew them on the background paper. There's no rule about what goes into the background and what moves around.

The mechanics of how the player jumps were really carefully designed. Simulating them in a paper prototype is a valuable learning exercise.

*Paper prototypes look like they'd be useful for **more than just games**. I bet I can use them in my other projects too.*

All of the tools and ideas in "Game Design...and Beyond" sections are important skills that go way beyond just game development—but we've found that they're easier to learn when you try them with games first.

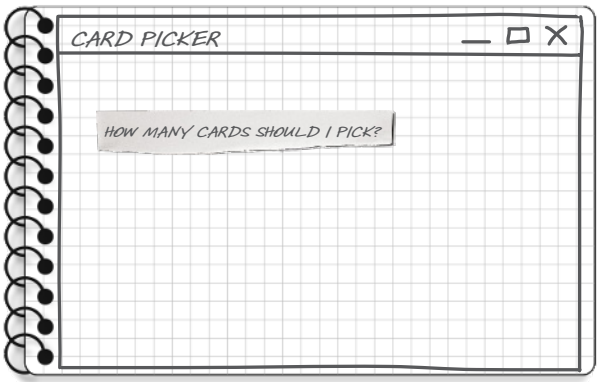
Yes! A paper prototype is a great first step for any project.

If you're building a desktop app, a mobile app, or any other project that has a user interface, building a paper prototype is a great way to get started. Sometimes you need to create a few paper prototypes before you get the hang of it. That's why we started with a paper prototype for a classic game…because that's a great way to learn how to build paper prototypes. **Prototyping is a really valuable skill for any kind of developer**, not just a game developer.

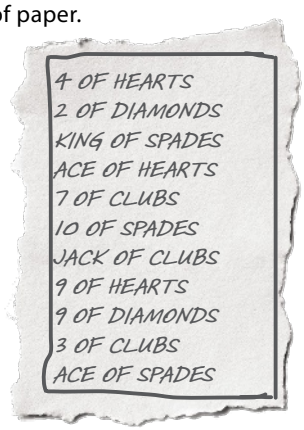
Sharpen your pencil

In the next project, you'll create a MAUI app that uses your CardPicker class to generate a set of random cards. In this paper-and-pencil exercise, you'll build a paper prototype of your app to try out various design options.

Start by drawing the window frame on a large piece of paper and a label on a smaller scrap of paper.



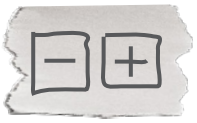
Your app needs to include a Button control with the text "Pick some cards" and a Label control to display the cards somewhere in the window.



Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There's no single right answer—there are lots of ways to design any app.

Your app needs a way for the user to choose the number of cards to pick. Try drawing an Entry control that they can use to type numbers into your app.

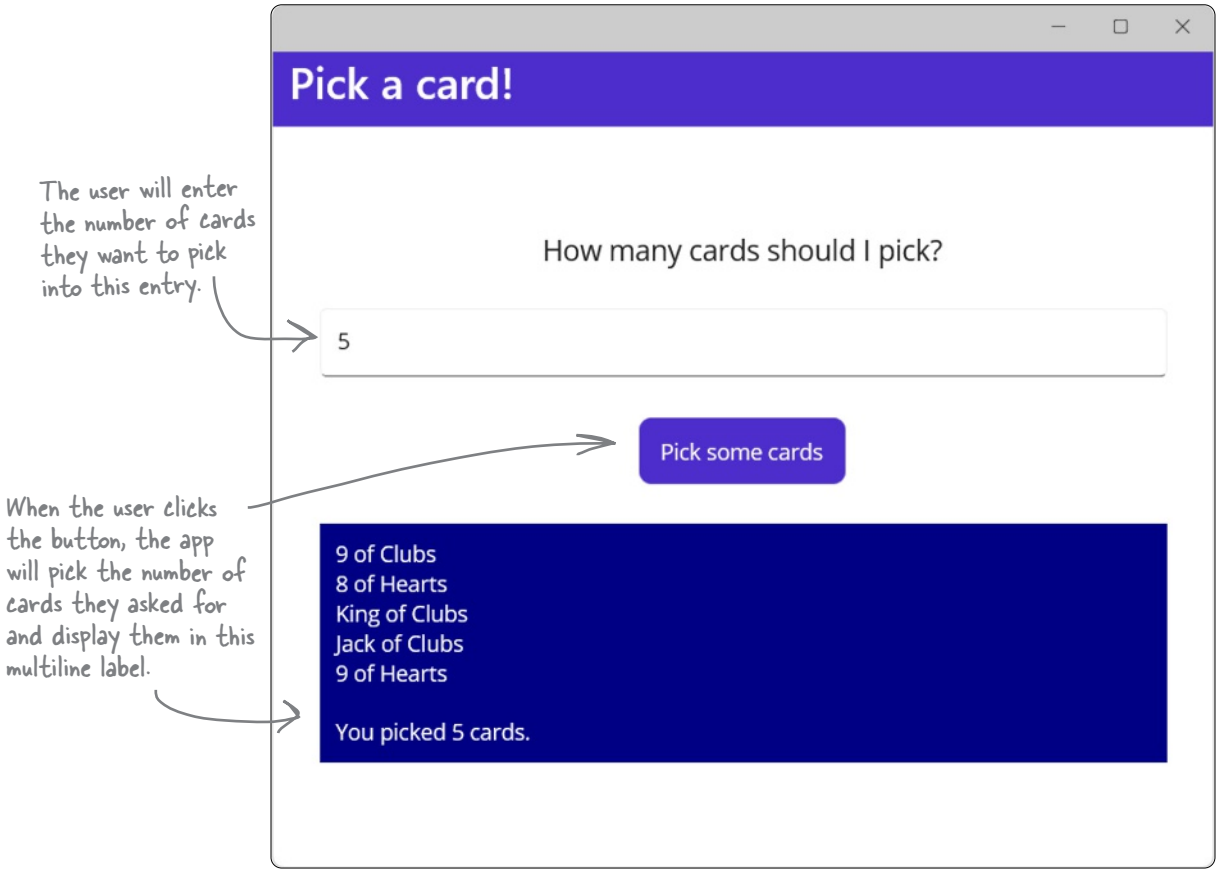
Try drawing Slider and Stepper controls too. Can you think of other controls that you've used to input numbers into apps before? Maybe a Picker? Get creative!



Build a MAUI version of your random card app

All of the code for picking random cards is conveniently organized into a class called CardPicker. Now you'll **reuse that class** in a .NET MAUI app.

Here's how the app will work.



Make your app accessible!

Accessibility is really important—and paying attention to accessibility is a great way to focus on important skills, like understanding your users and their needs:

- The Label and Entry controls each have a **SemanticProperties.Description property** so the screen reader will read it out loud.
- The Button control has a **SemanticProperties.Hint property** because the screen reader will read the contents of the button but we still want to give people who use accessibility tools additional context for the control.

Exercise

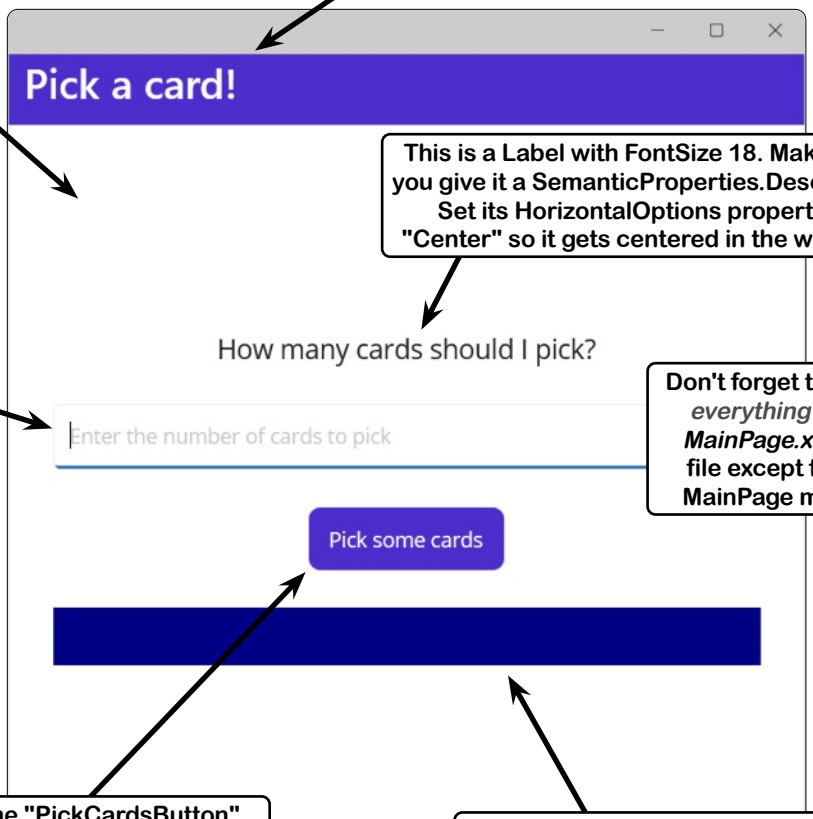
You already have the tools you need to create the XAML for the MAUI card picker app! In this exercise, you'll use what you learned about XAML in the first two chapters to create the main page for your app. You may need to go back to the XAML code you wrote in Chapter 2 to see how you added controls to your page.

Create a new .NET MAUI app called **PickRandomCardsMAUI**. Edit the *MainPage.xaml.cs* file to delete the controls inside the VerticalStackLayout (just like you did in Chapter 2), then add the controls for your card picker app.

Bonus: Edit the AppShell.xaml file to set the page title! We haven't shown you how to do that yet—can you figure it out?

Can you figure out how to set the page title? Open the *AppShell.xaml* file, look for a `<ShellContent>` tag, and change its Title property.

The ContentPage contains a ScrollView, which contains a VerticalStackLayout, just like your last MAUI project.



This is a Label with FontSize 18. Make sure you give it a SemanticProperties.Description. Set its HorizontalOptions property to "Center" so it gets centered in the window.

This is an Entry. Give it a placeholder and a semantic description for accessibility, and use the x:Name property to name it "NumberOfCards" so your code can read its value.

Don't forget to *delete everything* in the *MainPage.xaml.cs* file except for the MainPage method.

This Button has the name "PickCardsButton" and a *Clicked event handler method* called PickCardsButton_Clicked. Make sure the event handler method is created in *MainPage.xaml.cs*. Set its HorizontalOptions property to "Center" and give it a SemanticProperties.Hint property.

This is a multiline Label with the name "PickedCards" and a Padding property to 20 so it has some space around the text. It has white text on a dark blue background. Make sure it has a SemanticProperties.Description.

Peeking at the solution is *not cheating*! It's actually a great way to get these ideas to stick in your brain.

Exercise Solution

Your *MainPage.xaml.cs file* should have a public MainPage() method that calls InitializeComponent and an empty Clicked event handler method and nothing else.

Here's the XAML for the contents of *MainPage.xaml* (we didn't include the outer `<ContentPage>` tag):

```
<ScrollView>
    <VerticalStackLayout
        Spacing="25"
        Padding="30,0"
        VerticalOptions="Center">

        <Label
            Text="How many cards should I pick?"
            SemanticProperties.Description="How many cards should I pick?"
            FontSize="18"
            HorizontalOptions="Center" />

        <Entry
            x:Name="NumberOfCards"
            SemanticProperties.Description="Enter the number of cards to pick"
            Placeholder="Enter the number of cards to pick" />

        <Button
            x:Name="PickCardsButton"
            Text="Pick some cards"
            SemanticProperties.Hint="Picks some cards"
            Clicked="PickCardsButton_Clicked"
            HorizontalOptions="Center" />

        <Label x:Name="PickedCards" Padding="20"
               TextColor="White" BackgroundColor="DarkBlue"
               SemanticProperties.Description="Shows the cards that were picked" />

    </VerticalStackLayout>
</ScrollView>
```

These are the same ScrollView and VerticalStackLayout tags that Visual Studio created using the .NET MAUI template.

The HorizontalOptions property centers the label on the page. Try the other options—do you like the way they look better?

You gave the Entry, Button, and Label controls names that you'll use in your C# code.

Make sure Visual Studio added the PickCardsButton_Clicked *event handler method* that gets called when the button is clicked. You'll use it in the second part of this project. If you're using VSCode you might need to carefully type it yourself, just like you did with the other MAUI projects.

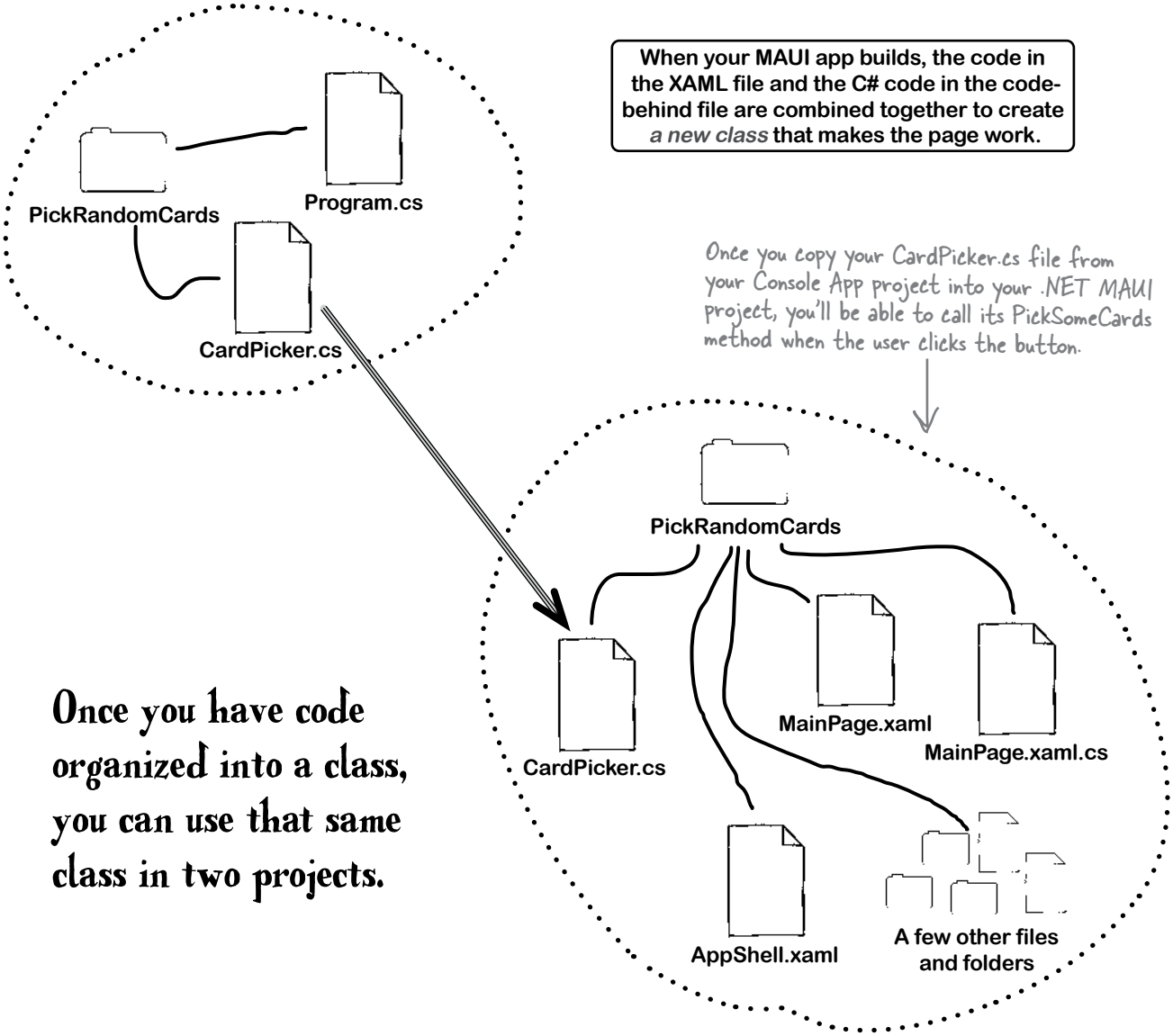
We made this change to AppShell.xaml to set the title of the page to "Pick a card!":

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    ...
    Shell.FlyoutBehavior="Disabled" BackgroundColor="Red">

    <ShellContent
        Title="Pick a card!"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />

</Shell>
```

The XAML in *AppShell.xaml* tells your MAUI app what to do when it first starts up. The ShellContent's Route property tells it to load the page in your *MainPage.xaml* file. Try setting the BackgroundColor of the outer `Shell` tag—what does that change in the app?

Make your MAUI app pick random cards

You've got an app that looks like it's supposed to, and that's a great start! In the second part of this project, you'll make it work, so when the user enters a number and clicks the button it picks random cards. That's where your CardPicker class comes in. You've already created a class that picks random cards. Now you just need to **copy that class into your new APP**. Once it's copied, you'll be able to make your button's event handler method call the PickSomeCards method in the CardPicker class.



Once you have code organized into a class, you can use that same class in two projects.

Reuse your CardPicker class

You took the time to put all of the random card picking code into a convenient class. Now it's time to take **reuse that class** by ***copying the file*** with the C# code into your new MAUI project.

Do this!

1 **Choose Add Existing Item in Visual Studio or manually copy the file in VSCode.**
This feature in the IDE will copy an existing file into your project. You created a file called *CardPicker.cs* in your PickRandomCards console app. Now you'll tell the IDE to **add that class file** to your MAUI project, which will cause it to copy the file into your MAUI app's project folder.

- In Visual Studio, right-click on the project in the Solution Explorer window and choose Add >> Existing Item (Shift+Alt+A), or choose Add Existing Item from the Project menu.
- In VSCode, you'll need to manually copy the file into the folder. Right-click on the project in the Solution Explorer and choose "Reveal in File Explorer" (or "Reveal in Finder" if you're using a Mac). Use your operating system to copy the file into your project folder that VSCode opened. Once the file is copied, it will automatically appear in the Solution Explorer.

2 **Find your *CardPicker.cs* file and add it to your project.**
The IDE will pop up a folder explorer window. Navigate to the folder with your PickACard console app and **double-click on CardPicker.cs**. You should now see CardPicker in the Solution Explorer.

Make sure CardPicker.cs now shows up in your Solution Explorer. Open it and make sure that you see the code for the CardPicker class from earlier in the chapter.

3 **Try to use your CardPicker class in the MainPage.xaml.cs code.**
Open *MainPage.xaml.cs*. Make sure you've deleted everything inside the MainPage class except the MainPage method (starting with `public MainPage()` and ending with a closing bracket). Edit the PickCardsButton_Clicked event handler method and try adding a statement that calls your CardPicker.PickSomeCards method.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PickCardsButton_Clicked(object sender, EventArgs e)
    {
        CardPicker.
    }
}
```

Here's the event handler method that Visual Studio added to your C# code when you added a Clicked event handler to the XAML for the button. If you're using VSCode, you'll need to carefully type the empty method into your MainPage.xaml.cs file.

Hold on—something's wrong!

When you start typing the statement to call CardPicker.PickSomeCards, Visual Studio doesn't pop up its normal IntelliSense window, and there's a squiggly error line under CardPicker.

Why do you think Visual Studio is treating CardPicker like that?

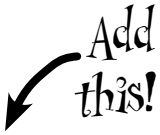
Add a using directive to use code in another namespace

You used either a **file-scoped namespace** or **block-scoped namespace** to put your CardPicker class in the PickRandomCards namespace. Compare the namespace declaration in your CardPicker class to the code at the top of your *MainPage.xaml.cs* file in your MAUI app:

```
namespace PickRandomCardsMAUI;

public partial class MainPage : ContentPage
{
  ... your MAUI app's code is in the PickRandomCardsMAUI namespace ...
}
```

The reason your MainPage class can't access the methods in your CardPicker class is because ***they're in different namespaces***.

Luckily, C# has an easy way to deal with this. You'll add a **`using` directive** in your code that calls the methods in CardPicker—that's a special line that you put at the top of a class file to tell it to use code in another namespace.



Add this line to the top of your MainPage.xaml.cs file.
If you chose a different name for your console app, replace PickRandomCards with the namespace in your *CardPicker.cs* file.

```
using PickRandomCards;
```

This *using directive* will let you add code to your *MainPage.xaml.cs* file that uses classes in the PickRandomCards namespace—so now you can write code that calls methods in your CardPicker class. You might see other using directives at the top of the file too.

Now go back to the event handler method for your button. Start typing `CardPicker.` like you did before. Now Visual Studio will pop up its IntelliSense window, just like you'd expect it to.

Exercise

Here's a C# coding challenge for you! Now that you added the `using` directive to the top of your *MainPage.xaml.cs* file, code in that file can use the CardPicker class. Can you finish your event handler method to make your app work?

To do this, you'll need to add statements to the PickCardsButton_Clicked event handler method. Here's what to do:

- The first thing the method does is call int.TryParse to convert NumberOfCards.Text to a number.
- If the number is valid, it calls CardPicker.PickSomeCards just like in your console app. If it isn't, it makes the PickedCards label display a message: `PickedCards.Text = "Please enter a valid number.";`
- Instead of writing to the console, it sets PickedCards.Text to a string value to make text appear in the PickedCards Label control. You can clear the text in PickedCards like this: `PickedCards.Text = String.Empty;`
- After it clears the PickedCards label, it uses a `foreach` loop that works just like the one in your console app.
- Add this statement after the `foreach` loop to tell the user how many cards they picked: `PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";`

Exercise Solution

Here's the finished event handler method.

```
private void PickCardsButton_Clicked(object sender, EventArgs e)
{
  if (int.TryParse(NumberOfCards.Text, out int numberOfCards))
  {
    string[] cards = CardPicker.PickSomeCards(numberOfCards);
    PickedCards.Text = String.Empty;
    foreach (string card in cards)
    {
      PickedCards.Text += card + Environment.NewLine;
    }
    PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";
  }
  else
  {
    PickedCards.Text = "Please enter a valid number.";
  }
}
```

Now that you have a using directive at the top of your MainPage.xaml.cs file, you can use the CardPicker class.

The foreach loop works just like the one in the console app, except instead of writing a line of text to the console it adds a line to the multi-line PickedCards Label control.

What happens if you don't add this last line to your PickedCards Label? Does it look weird? Can you sleuth out how to fix it?

Bullet Points

- Classes have methods that contain statements that perform actions. Well-designed classes have sensible method names.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts with the `int` keyword returns an int value. Here's a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. So if a method declaration has the string return type then you need a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts `public void` doesn't return anything at all. You can still use a `return` statement to exit a void method, as in this example: `if (finishedEarly) { return; }`
- Developers often **reuse** the same code in multiple programs. Classes can help you make your code more reusable.
- When you **select a control** in the XAML code editor, you can edit its properties in the Properties window.
- The XAML code combines with the C# code in the code-behind file to **create a new class**.
- You can create an array of values using a **collection expression** by putting the values between a pair of square brackets `[ ]` and separating them with commas.
- The **global namespace** is contains the top-level statements and any class not explicitly put into a namespace using a namespace declaration.

Ana's prototypes look great...

Ana found out that whether her player was being chased by an alien, a pirate, a zombie, or an evil killer clown, she could use the same methods from her Enemy class to make them work. Her game is starting to shape up.

Enemy
- SearchForPlayer
- SpottedPlayer
- CommunicatePlayerLocation
- StopSearching
- CapturePlayer

...but what if she wants more than one enemy?

And that's great…until Ana wants more than one enemy, which is all there was in each of her early prototypes. What should she do to add a second or third enemy to her game?

Ana *could* copy the Enemy class code and paste it into two more class files. Then her program could use methods to control three different enemies at once. Technically, we're reusing the code…right?

Hey Ana, what do you think of that idea?

She has a point. What if she wants a level with, say, dozens of zombies? Creating dozens of identical classes just isn't practical.

Enemy1
- SearchForPlayer
- SpottedPlayer
- CommunicatePlayerLocation
- StopSearching
- CapturePlayer

Enemy2
- SearchForPlayer
- SpottedPlayer
- CommunicatePlayerLocation
- StopSearching
- CapturePlayer

Enemy3
- SearchForPlayer
- SpottedPlayer
- CommunicatePlayerLocation
- StopSearching
- CapturePlayer

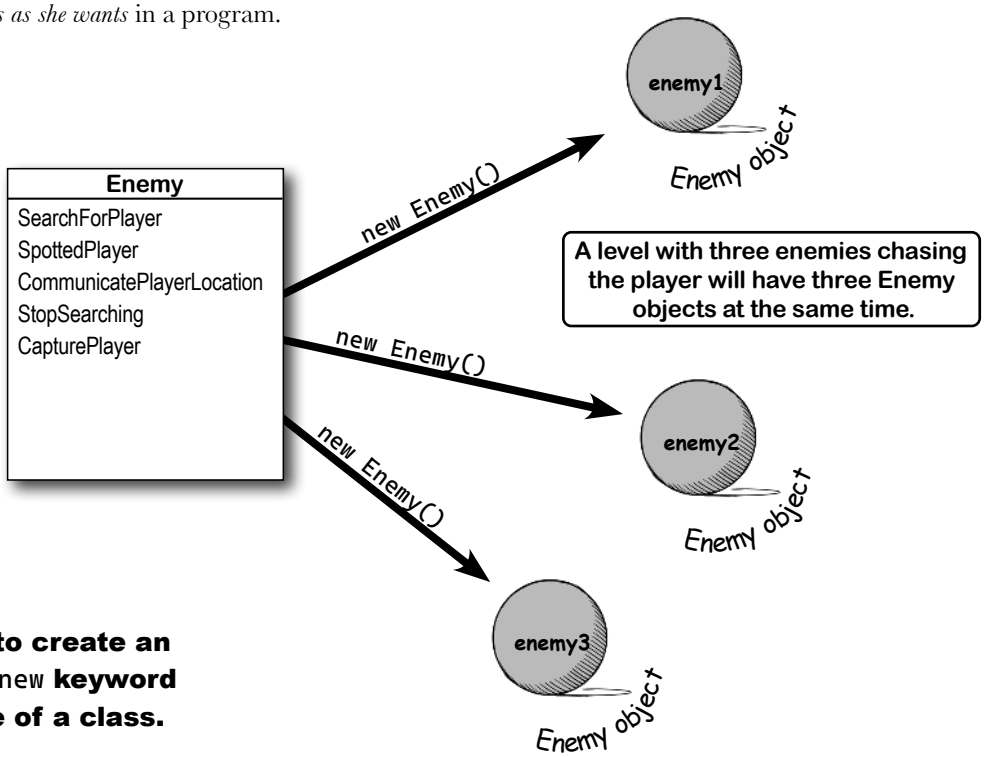
Are you joking? Using separate identical classes for each enemy is a **terrible idea**. What if I want more than three enemies at once?

Maintaining three copies of the same code is really messy.

A lot of problems you have to solve need a way to represent one ***thing*** a bunch of different times. In this case, it's an enemy in a game, but it could be songs in a music player app, or contacts in a social media app. Those all have one thing in common: they always need to treat the same kind of thing in the same way, no matter how many of that thing they're dealing with. Let's see if we can find a better solution.

Ana can use objects to solve her problem

Objects are C#'s tool that you use to work with a bunch of similar things. Ana can use objects to program her Enemy class just once, and use it *as many times as she wants* in a program.



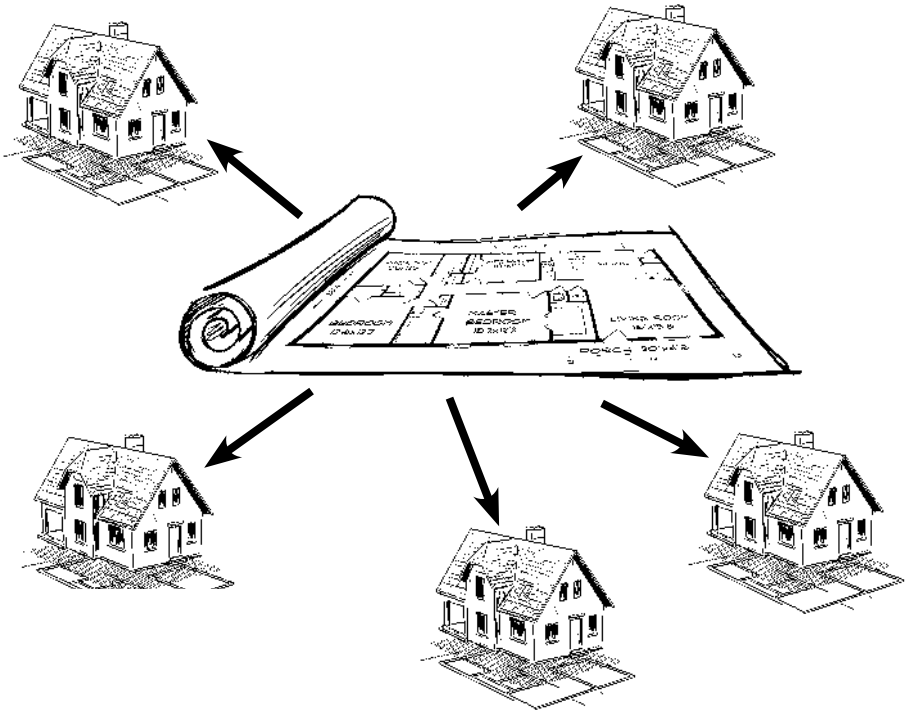
All you need to create an object is the `new` keyword and the name of a class.

```
Enemy enemy1 = new Enemy();
enemy1.SearchForPlayer();
if (enemy1.SpottedPlayer()) {
   enemy1.CommunicatePlayerLocation();
} else {
   enemy1.StopSearching();
}
```

Now you can use the object! When you create an object from a class, that object has all of the methods from that class.

You use a class to build an object

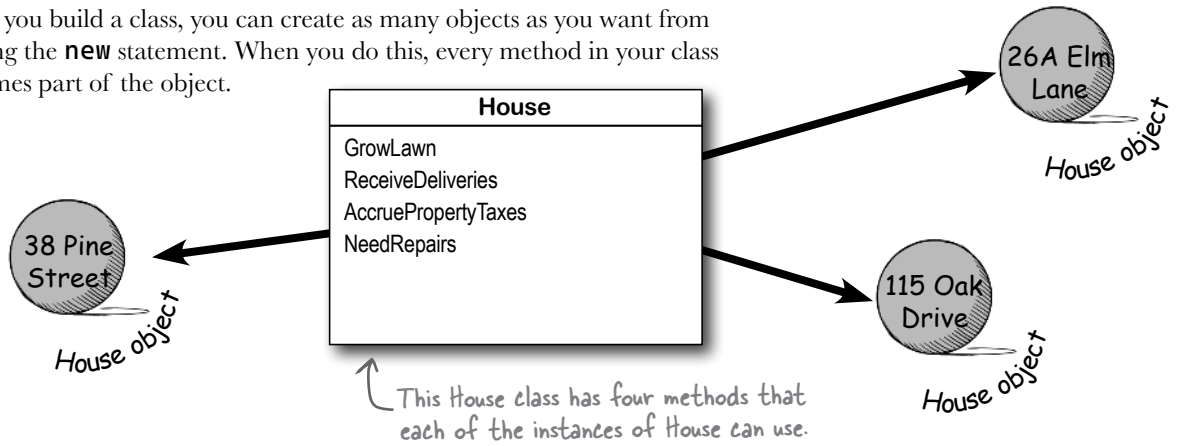
A class is like a blueprint for an object. If you wanted to build five identical houses in a suburban housing development, you wouldn't ask an architect to draw up five identical sets of blueprints. You'd just use one blueprint to build five houses.



A class defines its members, just like a blueprint defines the layout of the house. You can use one blueprint to make any number of houses, and you can use one class to make any number of objects.

An object gets its methods from its class

Once you build a class, you can create as many objects as you want from it using the `new` statement. When you do this, every method in your class becomes part of the object.



When you create a new object from a class, it's called an instance of that class

You use the **new keyword** to create an object. All you need is a variable to use with it. Use the class as the variable type to declare the variable, so instead of int or bool, you'll use a class like House or Enemy.

> in-stance, noun.
> an example or one occurrence of something. *The IDE search-and-replace feature finds every* ***instance*** *of a word and changes it to another.*

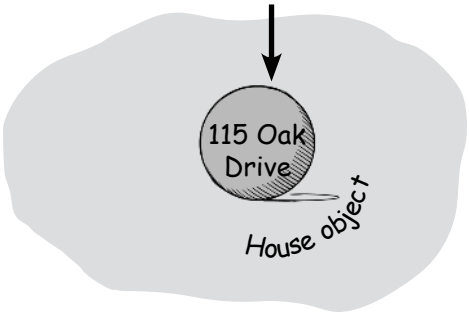
Before: here's a picture of your computer's memory when your program starts.

Your program executes a new statement.

```
House oakDrive115 = new House();
```

This "new" statement creates a new House object and assigns it to a variable called oakDrive115.

After: now it has an instance of the House class in memory.



That ***new keyword*** *looks familiar. I've seen this somewhere before, haven't I?*

Yes! You've already created instances in your own code.

Go back to your animal matching program and look for this line of code:

```
Random random = new Random();
```

You created an instance of the Random class, and then you called its Next method. Now look at your CardPicker class and find the **new** statement. You've been using objects this whole time!

A better solution for Ana...brought to you by objects

Ana used objects to reuse the code in the Enemy class without all that messy copying that would've left duplicate code all over her project. Here's how she did it.

1. Ana created a Level class that stored the enemies in an **Enemy array** called `enemyArray`, just like you used string arrays to store cards and animal emoji.

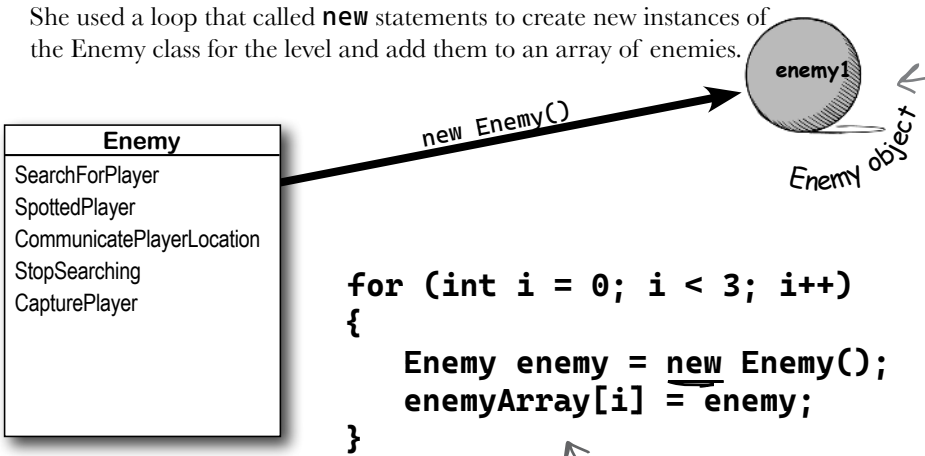
```
internal class Level {
    Enemy[] enemyArray = new Enemy[3];
```

Hmm, this array is inside the class, but outside of the methods. What do you think is going on?

Use the name of a class to declare an array of instances of that class.

We're using the "new" keyword to create an array of Enemy objects, just like you did earlier with strings.

2. She used a loop that called `new` statements to create new instances of the Enemy class for the level and add them to an array of enemies.



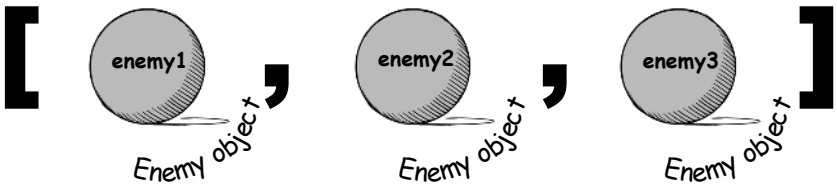
The enemy1 object is an instance of the Enemy class.

```
for (int i = 0; i < 3; i++)
{
    Enemy enemy = new Enemy();
    enemyArray[i] = enemy;
}
```

This statement uses the new keyword to create an Enemy object.

This statement adds the newly created Enemy object to the array.

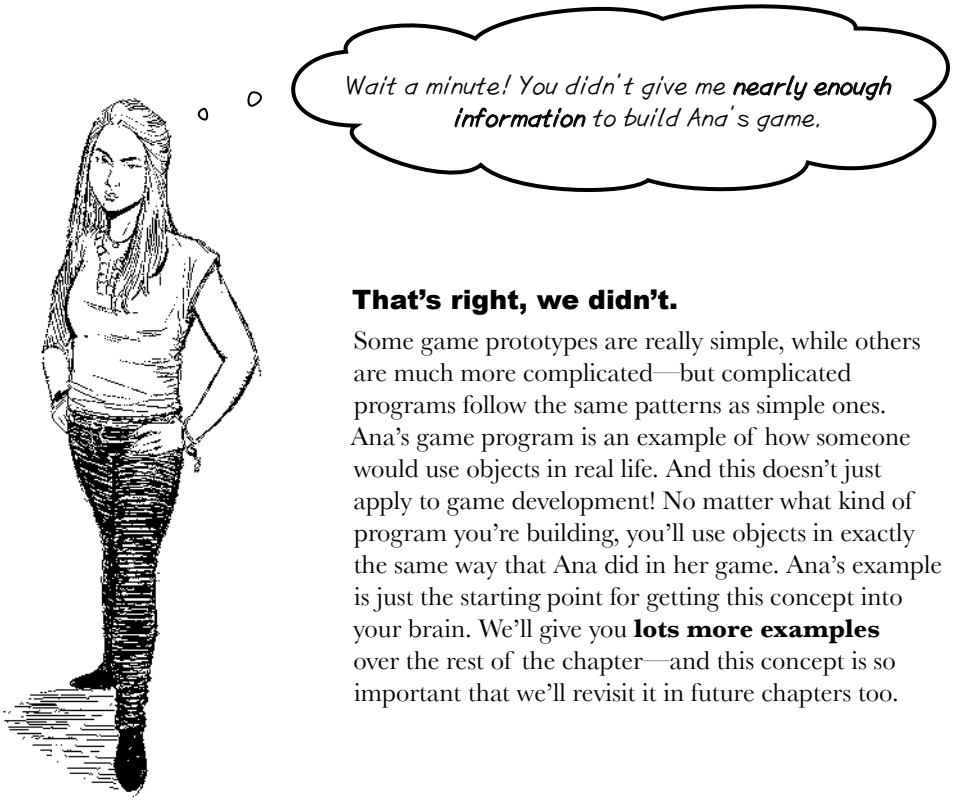
3. She called methods of each Enemy instance during every frame update to implement the enemy behavior.




```
foreach (Enemy enemy in enemyArray)
{
    // code that calls the Enemy methods
}
```

The foreach loop iterates through the array of Enemy objects.

When you create a new instance of a class, it's called *instantiating* that class.



That's right, we didn't.

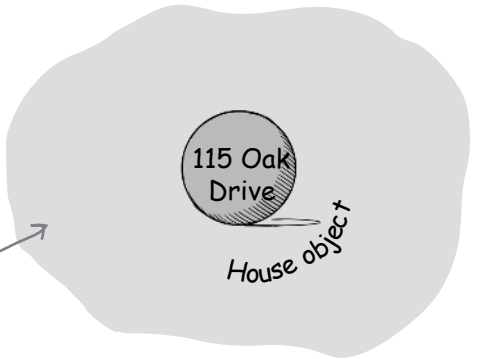
Some game prototypes are really simple, while others are much more complicated—but complicated programs follow the same patterns as simple ones. Ana's game program is an example of how someone would use objects in real life. And this doesn't just apply to game development! No matter what kind of program you're building, you'll use objects in exactly the same way that Ana did in her game. Ana's example is just the starting point for getting this concept into your brain. We'll give you **lots more examples** over the rest of the chapter—and this concept is so important that we'll revisit it in future chapters too.

Theory and practice

Speaking of patterns, here's a pattern that you'll see over and over again throughout the book. We'll introduce a concept or idea (like objects) over the course of a few pages, using pictures and short code excerpts to demonstrate the idea. This is your opportunity to take a step back and try to understand what's going on without having to worry about getting a program to work.

```
House oakDrive115 = new House();
```

When we're introducing a new concept (like objects), keep your eyes open for pictures and code excerpts like these.



Sharpen your pencil

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Open any Console App project that uses top-level statements (or create a new one). Press Enter to start a new statement, then type `Random.Shared.`—as soon as you type the second period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (). We filled in some of the methods. Finish filling in the class diagram for the Random class.

Random
Equals GetHashCode GetType .... ToString

In Chapter 2, we showed you three types, int (for whole numbers), string (for text), and bool (for true/false values). A *double* is another type that's used for numbers with decimal places. The computer science term for a number with decimal places is a *floating-point number*. You'll learn about more types in the next chapter.

2. Write code to create a new array of doubles called **`randomDoubles`**, then use a `for` loop to add 20 double values to that array. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code—make sure you're calling the method that returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0. (We'll talk about what "floating point" means in the next chapter.)

```
double[] randomDoubles = new double[20];

{
    double value =

}
```

We filled in part of the code, including the curly braces. Your job is to finish those statements and then write the rest of the code.

Sharpen your pencil Solution

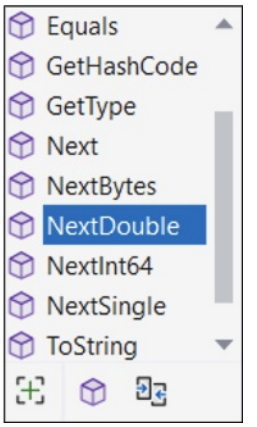
Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Open any Console App project that uses top-level statements (or create a new one). Press Enter to start a new statement, then type `Random.Shared.`—as soon as you type the second period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (). We filled in some of the methods. Finish filling in the class diagram for the Random class.

Random
Equals
GetHashCode
GetType
Next
NextBytes
NextDouble
NextInt64
NextSingle
ToString

`Random.Shared.`

Here's the IntelliSense window that Visual Studio popped up when you typed Shared.Random.



When you select NextDouble in the IntelliSense window, it shows documentation for the method.

double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.

2. Write code to create a new array of doubles called **`randomDoubles`**, then use a **`for`** loop to add 20 double values to that array. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code—make sure you're calling the method that returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0. (We'll talk about what "floating point" means in the next chapter.)

```
double[] randomDoubles = new double[20];
for (int i = 0; i < 20; i++)
{
    double value = Random.Shared.NextDouble();
    randomDoubles[i] = value;
}
```

This is really similar to the code that you used in your CardPicker class.

An instance uses fields to keep track of things

You've seen how classes can contain fields as well as methods. We just saw how you used the `static` keyword to declare a field in your CardPicker class:

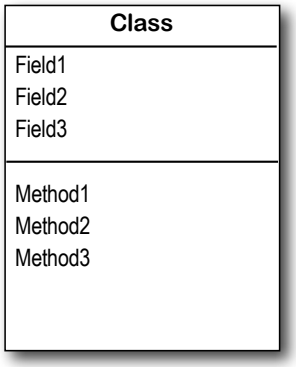
```
static Random random = new Random();
```

What happens if you take away that `static` keyword? Then the field becomes an **instance field**, and every time you ***instantiate*** the class, the new instance that was created *gets its own copy* of that field.

Sometimes people think the word "instantiate" sounds a little weird, but it makes sense when you think about what it means: creating a new instance of a class.

When we want to include fields in a class diagram, we'll draw a horizontal line in the box. The fields go above the line, and methods go below the line.

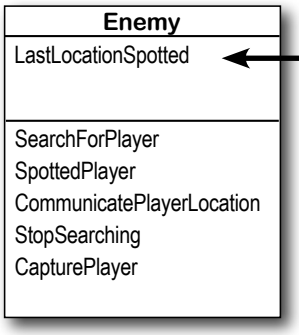
This is where a class diagram shows the fields. Every instance of the class uses its own copy of each field to keep track of its state.



Class diagrams typically list all of the fields and methods in the class. We call them the *class members*.

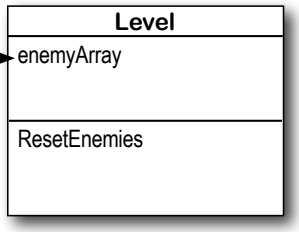
Methods are what an object does. Fields are what an object knows.

When Ana's prototype created three instances of her Enemy class, each of those objects was used to keep track of a different enemy in the game. Every instance keeps separate copies of the same data: setting a field on the enemy2 instance won't have any effect on the enemy1 or enemy3 instances.



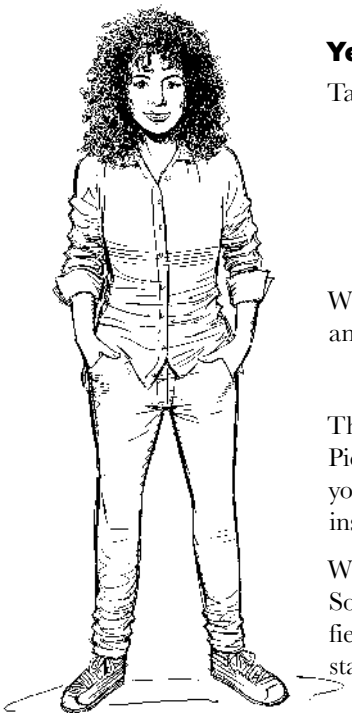
Each enemy in Ana's game uses a field to keep track of the last location where it spotted the player.

Remember how the Level class used an array to keep track of the Enemy objects? That was a field!



An object's behavior is defined by its methods, and it uses fields to keep track of its state.

*I used the **new** keyword to create an instance of Random, but I **never created a new instance** of my CardPicker class. So does that mean I can call methods without creating objects?*



Yes! That's why you used the `static` keyword in your declarations.

Take another look at the method declarations in your CardPicker class:

```
public static string[] PickSomeCards(int numberOfCards)

private static string RandomValue()

private static string RandomSuit()
```

When you use the **`static`** keyword to declare a field or method in a class, you don't need an instance of that class to access it. You just called your method like this:

```
CardPicker.PickSomeCards(numberOfCards)
```

That's how you call static methods. If you take away the **`static`** keyword from the PickSomeCards method declaration, then you'll have to create an instance of CardPicker you can use to call the method. Other than that distinction, static methods are just like instance methods: they can take arguments, they can return values, and they live in classes.

When a field is static **there's only one copy of it, and it's shared by all instances**. So if you created multiple instances of CardPicker, they would all share the same *random* field. You can even mark your **whole class** as static, and then all of its members **must** be static too. If you try to add a nonstatic method to a static class, your program won't build.

there are no Dumb Questions

Q: When I think of something that's "static," I think of something that doesn't change. Does that mean nonstatic methods can change, but static methods don't? Do they behave differently?

A: No, both static and nonstatic methods act exactly the same. The only difference is that static methods don't require an instance, while nonstatic methods do.

Q: So I can't use my class until I create an instance of an object?

A: You can use its static methods, but if you have methods that aren't static, then you need an instance before you can use them.

Q: Then why would I want a method that needs an instance? Why wouldn't I make all my methods static?

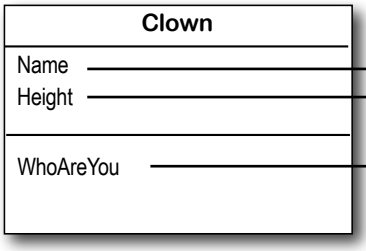
A: Because if you have an object that's keeping track of certain data—like Ana's instances of her Enemy class that each kept track of different enemies in her game—then you can use each instance's methods to work with that data. So when Ana's game calls the StopSearching method on the enemy2 instance, it only causes that one enemy to stop searching for the player. It doesn't affect the enemy1 or enemy3 objects, and they can keep searching. That's how Ana can create game prototypes with any number of enemies, and her programs can keep track of all of them at once.

When a field is static, there's only one copy of it shared by all instances.

Sharpen your pencil

Here's a console app that uses top-level statements and writes several lines to the console. It includes a class called Clown that has two fields, Name and Height, and a method called WhoAreYou that uses those fields to write a line to the console. Your job is to read the code and **write down the lines that are printed to the console**.

Here's the class diagram and code for the Clown class:




```
class Clown {
    public string? Name;
    public int Height;

    public void WhoAreYou() {
        Console.WriteLine("My name is " + Name +
            " and I'm " + Height + " inches tall.");
    }
}
```

Here are the contents of the *Program.cs* file. There are comments next to each of the calls to the WhoAreYou method, which prints a line to the console. Your job is to fill in the blanks in the comments so they match the output.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();      // My name is _______ and I'm ____ inches tall.

Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();  // My name is _______ and I'm ____ inches tall.

Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();        // My name is _______ and I'm ____ inches tall.

anotherClown.Height *= 2;
anotherClown.WhoAreYou();  // My name is _______ and I'm ____ inches tall.
```

The *= operator tells C# to take whatever's on the left of the operator and multiply it by whatever's on the right, so this will update the Height field.

Thanks for the memory

When your program creates an object, it lives in a part of the computer's memory called the **heap**. When your code creates an object with a `new` statement, C# immediately reserves space in the heap so it can store the data for that object.

Here's a picture of the heap before the project starts. Notice that it's empty.

When your program creates a new object, it gets added to the heap.

Sharpen your pencil Solution

Here's what the program prints to the console. It's worth taking a few minutes to create a new console app—make sure it uses top-level statements—add the Clown class, and make its *Program.cs* method the code below. Then step through it with the debugger so you can see exactly how it works.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();      // My name is Boffo and I'm 14 inches tall.

Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();  // My name is Biff and I'm 16 inches tall.

Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();        // My name is Biff and I'm 11 inches tall.

anotherClown.Height *= 2;
anotherClown.WhoAreYou();  // My name is Biff and I'm 32 inches tall.
```

What's on your app's mind

Let's take a closer look at the program in the "Sharpen your pencil" exercise, starting with the first line of the app. It's actually **two statements** combined into one:

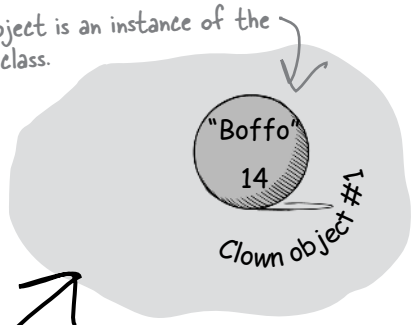
```
Clown oneClown = new Clown();
```

This is a statement that declares a variable called oneClown of type Clown.

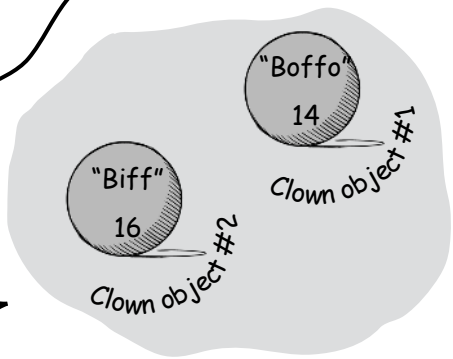
This statement creates a new object and assigns it to the oneClown variable.

Next, let's look closely at what the heap looks like after each group of statements is executed:

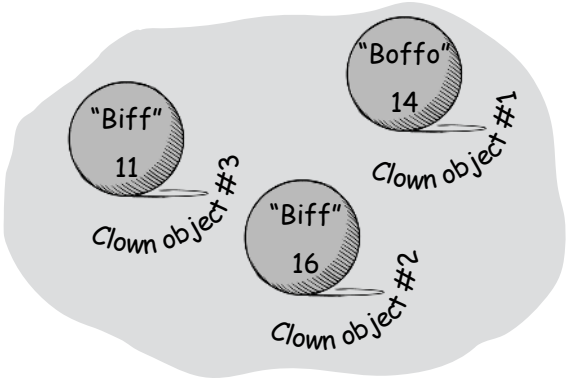
```
// These statements create an instance of
// Clown and then set its fields
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();
```



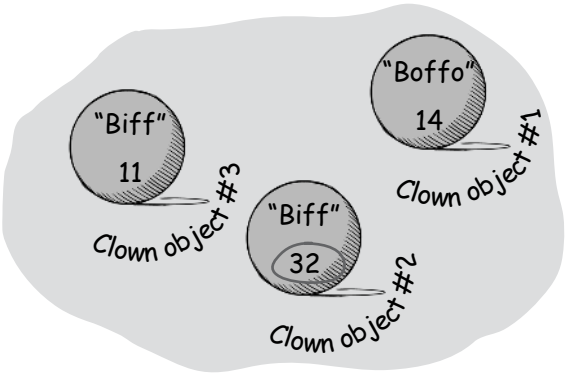

```
// These statements instantiate a second
// Clown object and fill it with data
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();
```




```
// Now we instantiate a third Clown object
// and use data from the other two
// instances to set its fields
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();
```




```
// Notice how there's no "new" statement
// here -- we're not creating a new object,
// just modifying one already in memory
anotherClown.Height *= 2;
anotherClown.WhoAreYou();
```



Sometimes code can be difficult to read

You may not realize it, but you're constantly making choices about how to structure your code. Do you use one method to do something? Do you split it into more than one? Do you even need a new method at all? The choices you make about methods can make your code much more intuitive—or if you're not careful, much more convoluted.

Here's a nice, compact chunk of code from a control program that runs a machine that makes candy bars:

```
int t = m.chkTemp();
if (t > 160) {
   T tb = new T();
   tb.clsTrpV(2);
   ics.Fill();
   ics.Vent();
   m.airsyschk();
}
```

Extremely compact code can be especially problematic

Take a second and look at that code. Can you figure out what it does? Don't feel bad if you can't—it's very difficult to read! Here are a few reasons why:

- We can see a few variable names: `tb`, `ics`, `m`. These are terrible names! We have no idea what they do. And what's that T class for?
- The chkTemp method returns an integer…but what does it do? We can guess maybe it has something to do with checking the temperature of…something?
- The clsTrpV method has one parameter. Do we know what that parameter is supposed to be? Why is it 2? What is that 160 number for?

*C# code in **industrial equipment**?! Isn't C# just for desktop apps, business systems, websites, and games?*

C# and .NET are everywhere...and we mean *everywhere*.

Have you ever played with a Raspberry PI? It's a low-cost computer on a single board, and computers like it can be found inside all sorts of machinery. Thanks to Windows IoT (or Internet of Things), your C# code can run on them.

You can learn more about .NET IoT apps here: *https://dotnet.microsoft.com/apps/iot*

Microsoft even has a free Raspberry PI simulator that you can use to get started:

https://learn.microsoft.com/azure/iot-hub/iot-hub-raspberry-pi-web-simulator-get-started

Most code doesn't come with a manual

Those statements don't give you any hints about why the code's doing what it's doing. In this case, the programmer was happy with the results because she was able to get it all into one method. But making your code as compact as possible isn't really useful! Let's break it up into methods to make it easier to read, and make sure the classes are given names that make sense.

We'll start by figuring out what the code is supposed to do. Luckily, we happen to know that this code is part of an **embedded system**, or a controller that's part of a larger electrical or mechanical system. And we happen to have documentation for this code—specifically, the manual that the programmers used when they originally built the system.

General Electronics Type 5 Candy Bar Maker Manual

The nougat temperature must be checked every 3 minutes by an automated system. If the temperature **exceeds 160°C**, the candy is too hot, and the system must **perform the candy isolation cooling system (CICS) vent procedure**:

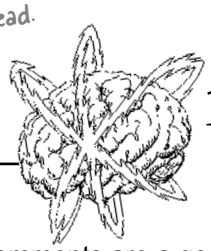
- Close the trip throttle valve on turbine #2.
- Fill the isolation cooling system with a solid stream of water.
- Vent the water.
- Initiate the automated check for air in the system.

How do you figure out what your code is supposed to do? Well, all code is written for a reason. So it's up to you to figure out that reason! In this case, we got lucky—we could look up the page in the manual that the developer followed.

We can compare the code with the manual that tells us what the code is supposed to do. Adding comments can definitely help us understand what it's supposed to do:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
   // Get the controller system for the turbines
   T tb = new T();

   // Close throttle valve on turbine #2
   tb.clsTrpV(2);

   // Fill and vent the isolation cooling system
   ics.Fill();
   ics.Vent();

   // Initiate the air system check
   m.airsyschk();
}
```

Adding extra line breaks to your code in some places can make it easier to read.



Brain Power

Code comments are a good start. Can you think of a way to make this code even easier to understand?

Use intuitive class and method names

That page from the manual made it a lot easier to understand the code. It also gave us some great hints about how to make our code easier to understand. Let's take a look at the first two lines:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
```

The comment we added explains a lot. Now we know why the conditional test checks the variable **t** against 160—the manual says that any temperature above 160°C means the nougat is too hot. It turns out that **m** is a class that controls the candy maker, with static methods to check the nougat temperature and check the air system.

So let's put the temperature check into a method, and choose names for the class and the methods that make their purpose obvious. We'll move these first two lines into their own method that returns a Boolean value, true if the nougat is too hot or false if it's OK:

```
/// <summary>
/// If the nougat temperature exceeds 160C it's too hot.
/// </summary>
public bool IsNougatTooHot() {
   int temp = CandyBarMaker.CheckNougatTemperature();
   if (temp > 160) {
     return true;
   } else {
     return false;
   }
}
```

When we rename the class "CandyBarMaker" and the method "CheckNougatTemperature," it starts to make the code easier to understand.

Notice how the C in CandyBarMaker is uppercase? If we always start class names with an uppercase letter and variables with lowercase ones, it's easier to tell when you're calling a static method vs. using an instance.

Did you notice the special `///` comments above the method? That's called an **XML Documentation Comment** (or **XMLDoc**) The IDE uses those comments to show you documentation for methods—like the documentation you saw when you used the IntelliSense window to figure out which method from the Random class to use.

IDE Tip: XML documentation for methods and fields

Visual Studio helps you add XML documentation. Put your cursor in the line above any method and type three slashes, and it will add an empty template for your documentation. If your method has parameters and a return type, it will add `<param>` and `<returns>` tags for them as well. Try going back to your CardPicker class and typing `///` in the line above the PickSomeCards method—the IDE will add blank XML documentation. Fill it in and watch it show up in IntelliSense.

```
/// <summary>
/// Picks a number of cards and returns them.
/// </summary>
/// <param name="numberOfCards">The number of cards to pick.</param>
/// <returns>An array of strings that contain the card names.</returns>
```

You can create XML documentation for your fields too. Try it out by going to the line just above any field and typing three slashes in the IDE. Anything you put after `<summary>` will show up in the IntelliSense window for the field.

What does the manual say to do if the nougat is too hot? It tells us to perform the candy isolation cooling system (or CICS) vent procedure. So let's make another method, and choose an obvious name for the T class (which turns out to control the turbine) and the ics class (which controls the isolation cooling system, and has two static methods to fill and vent the system), and cap it all off with some brief XML documentation:

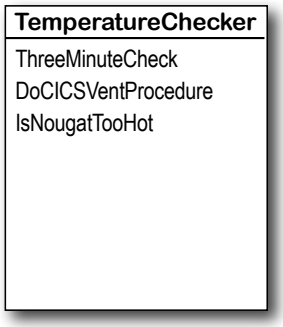
```
/// <summary>
/// Perform the Candy Isolation Cooling System (CICS) vent procedure.
/// </summary>
public void DoCICSVentProcedure() {
   TurbineController turbines = new TurbineController();
   turbines.CloseTripValve(2);
   IsolationCoolingSystem.Fill();
   IsolationCoolingSystem.Vent();
   Maker.CheckAirSystem();
}
```

When your method is declared with a void return type, that means it doesn't return a value and it doesn't need a `return` statement. All of the methods you wrote in Chapter 2 used the `void` keyword!

Now that we have the IsNougatTooHot and DoCICSVentProcedure methods, we can ***rewrite the original confusing code as a single method***—and we can give it a name that makes clear exactly what it does:

```
/// <summary>
/// This code runs every 3 minutes to check the temperature.
/// If it exceeds 160C we need to vent the cooling system.
/// </summary>
public void ThreeMinuteCheck() {
   if (IsNougatTooHot() == true) {
      DoCICSVentProcedure();
   }
}
```

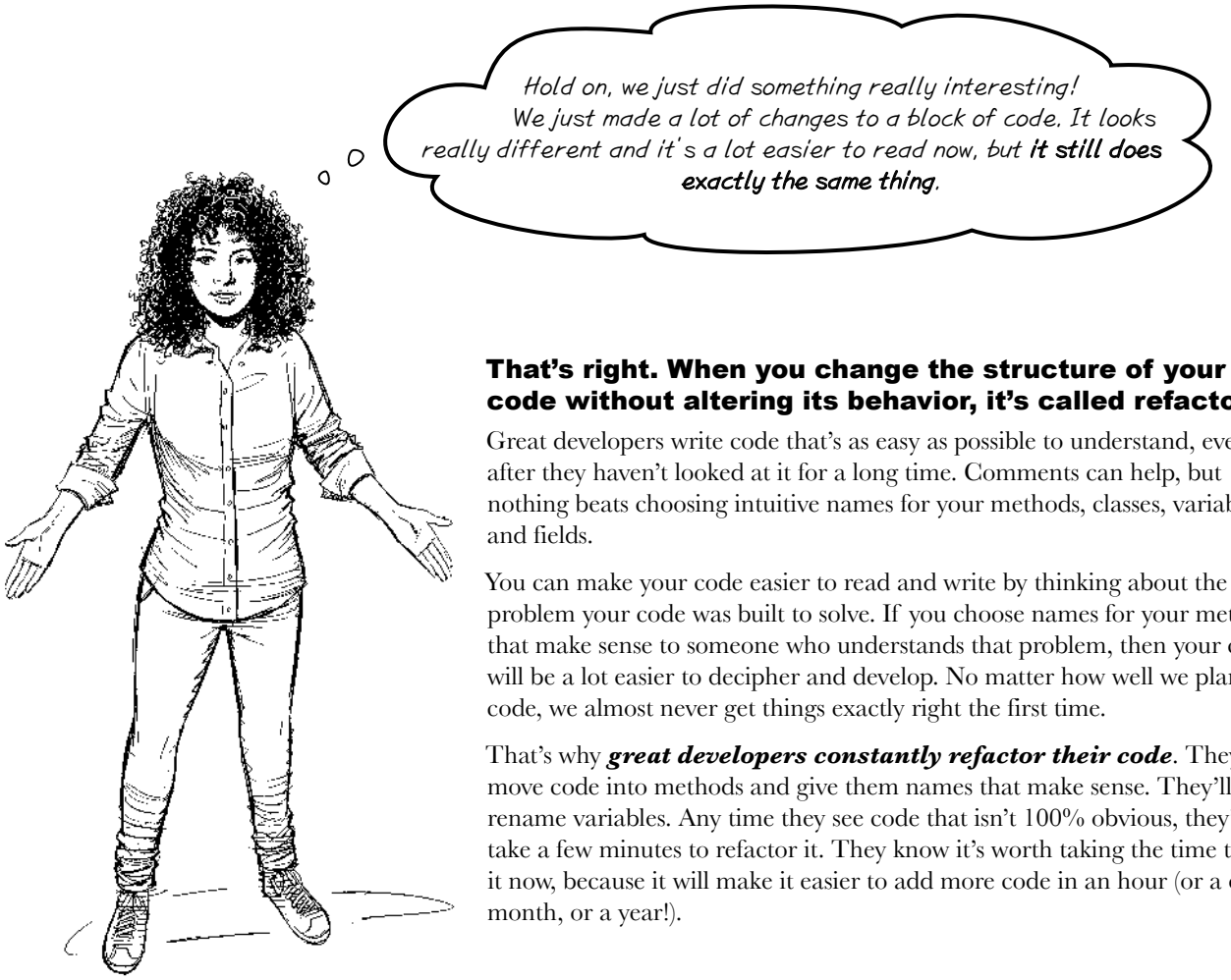
We bundled these new methods into a class called TemperatureChecker. Here's its class diagram.



Now the code is a lot more intuitive! Even if you don't know that the CICS vent procedure needs to be run if the nougat is too hot, **it's a lot more obvious what this code is doing**.

Use class diagrams to plan out your classes

A class diagram is a valuable tool for designing your code BEFORE you start writing it. Write the name of the class at the top of the diagram. Then write each method in the box at the bottom. Now you can see all of the parts of the class at a glance—and that's your first chance to spot problems that might make your code difficult to use or understand later.



That's right. When you change the structure of your code without altering its behavior, it's called refactoring.

Great developers write code that's as easy as possible to understand, even after they haven't looked at it for a long time. Comments can help, but nothing beats choosing intuitive names for your methods, classes, variables, and fields.

You can make your code easier to read and write by thinking about the problem your code was built to solve. If you choose names for your methods that make sense to someone who understands that problem, then your code will be a lot easier to decipher and develop. No matter how well we plan our code, we almost never get things exactly right the first time.

That's why ***great developers constantly refactor their code***. They'll move code into methods and give them names that make sense. They'll rename variables. Any time they see code that isn't 100% obvious, they'll take a few minutes to refactor it. They know it's worth taking the time to do it now, because it will make it easier to add more code in an hour (or a day, a month, or a year!).

Sharpen your pencil

Each of these classes has a serious design flaw. Write down what you think is wrong with each class, and how you'd fix it.

Class23

CandyBarWeight
PrintWrapper
GenerateReport
Go

This class is part of the candy manufacturing system from earlier.

.....

.....

.....

.....

DeliveryGuy

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

DeliveryGirl

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

These two classes are part of a system that a pizza parlor uses to track the pizza orders that are out for delivery.

.....

.....

.....

.....

CashRegister

MakeSale
NoSale
PumpGas
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

The CashRegister class is part of a program that's used by an automated convenience store checkout system.

.....

.....

.....

.....

Sharpen your pencil Solution

Here's how we improved the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they'll be used.

This class is part of the candy manufacturing system from earlier.

The class name doesn't describe what the class does. A programmer who sees a line of code that calls Class23.Go will have no idea what that line does. We'd also rename the method to something that's more descriptive—we chose MakeTheCandy, but it could be anything.

CandyMaker

CandyBarWeight
PrintWrapper
GenerateReport
MakeTheCandy

These two classes are part of a system that a pizza parlor uses to track the pizza orders that are out for delivery.

It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing—they represent a delivery person who's out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.

DeliveryPerson

~~Gender~~

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

We decided NOT to add a Gender field because there's actually no reason for this pizza delivery class to keep track of the gender of the people delivering pizza—and we should respect their privacy! Always look out for ways that bias can sneak into your code.

The CashRegister class is part of a program that's used by an automated convenience store checkout system.

All of the methods in the class do stuff that has to do with a cash register—making a sale, getting a list of transactions, adding cash...except for one: pumping gas. It's a good idea to pull that method out and stick it in another class.

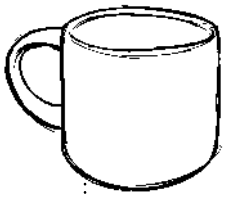
CashRegister

MakeSale
NoSale
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

Code Tip: A few ideas for designing intuitive classes

We're about to jump back into writing code. You'll be writing code for the rest of this chapter, and a LOT of code throughout the book. That means you'll be ***creating a lot of classes***. Here are a few things to keep in mind when you make choices about how to design them:

- **You're building your program to solve a problem.**
  Spend some time thinking about that problem. Does it break down into pieces easily? How would you explain that problem to someone else? These are good things to think about when designing your classes.
- **What real-world things will your program use?**
  A program to help a zookeeper track her animals' feeding schedules might have classes for different kinds of food and types of animals.
- **Use descriptive names for classes and methods.**
  Someone should be able to figure out what your classes and methods do just by looking at their names.
- **Look for similarities between classes.**
  Sometimes two classes can be combined into one if they're really similar. The candy manufacturing system might have three or four turbines, but there's only one method for closing the trip valve that takes the turbine number as a parameter.



Relax

It's OK if you get stuck when you're writing code. In fact, getting stuck can be a good thing!

Writing code is all about solving problems—and some of them can be tricky! But if you keep a few things in mind, it'll make the code exercises go more smoothly:

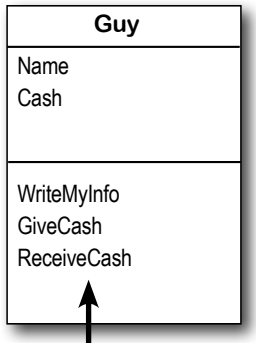
- It's easy to get caught up in syntax problems, like missing parentheses or quotes. One missing bracket can cause many build errors.
- It's **much better** to look at the solution than to get frustrated with a problem. When you're frustrated, your brain doesn't like to learn.
- All of the code in this book is tested and definitely works! But it's easy to accidentally type things wrong (like typing a one instead of a lowercase L, or missing a comma or semicolon).
- If your solution just won't build, try downloading it from the GitHub repository for the book—it has working code for everything in the book: *https://github.com/head-first-csharp/fifth-edition*

You can learn a lot from reading code. So if you run into a problem with a coding exercise, don't be afraid to peek at the solution. *It's* not *cheating!*

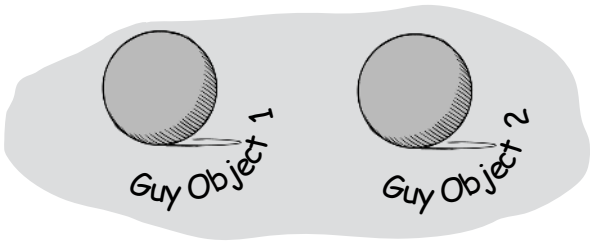
Build a class to work with some guys

Joe and Bob lend each other money all the time. Let's create a class to keep track of how much cash they each have. We'll start with an overview of what we'll build.

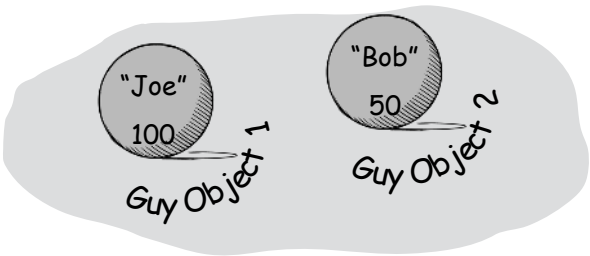
A guy keeps cash ("bucks") in his wallet so he can spend it on stuff. He can give cash to another guy, or receive cash from another guy.



1 **We'll create two instances of a "Guy" class.**
We'll use two Guy variables called `joe` and `bob` to keep track of each of our instances. Here's what the heap will look like after they're created:



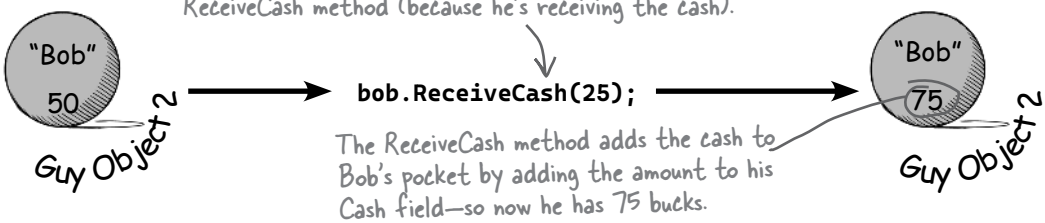
2 **We'll set each Guy object's Cash and Name fields.**
The two objects represent different guys, each with his own name and a different amount of cash in his pocket. Each guy has a Name field that keeps track of his name, and a Cash field that has the number of bucks in his pocket.



We chose names for the methods that make sense. You call a Guy object's GiveCash method to make him give up some of his cash, and his ReceiveCash method when you want to give cash to him (so he receives it).

3 **We'll add methods to give and receive cash.**
We'll make a guy give cash from his pocket (and reduce his Cash field) by calling his GiveCash method, which will return the amount of cash he gave. We'll make him receive cash and add it to his pocket (increasing his Cash field) by calling his ReceiveCash method.

If we want to give Bob 25 bucks, we call his ReceiveCash method (because he's receiving the cash).



The ReceiveCash method adds the cash to Bob's pocket by adding the amount to his Cash field—so now he has 75 bucks.

```
internal class Guy
{
    public string? Name;
    public int Cash;

    /// <summary>
    /// Writes my name and the amount of cash I have to the console.
    /// </summary>
    public void WriteMyInfo()
    {
        Console.WriteLine(Name + " has " + Cash + " bucks.");
    }

    /// <summary>
    /// Gives some of my cash, removing it from my wallet (or printing
    /// a message to the console if I don't have enough cash).
    /// </summary>
    /// <param name="amount">Amount of cash to give.</param>
    /// <returns>
    /// The amount of cash removed from my wallet, or 0 if I don't
    /// have enough cash (or if the amount is invalid).
    /// </returns>
    public int GiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't a valid amount");
            return 0;
        }
        if (amount > Cash)
        {
            Console.WriteLine(Name + " says: " +
                "I don't have enough cash to give you " + amount);
            return 0;
        }
        Cash -= amount;
        return amount;
    }

    /// <summary>
    /// Receive some cash, adding it to my wallet (or printing
    /// a message to the console if the amount is invalid).
    /// </summary>
    /// <param name="amount">Amount of cash to receive.</param>
    public void ReceiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't an amount I'll take");
        }
        else
        {
            Cash += amount;
        }
    }
}
```

The Name and Cash fields keep track of the guy's name and how much cash he has in his pocket. Don't forget the question mark when you declare the **string?** field. We'll talk more about what that's about in the next chapter.

Sometimes you want to ask an object to perform a task, like printing a description of itself to the console.

Don't add this Guy class to an app yet—just read it. Can you figure out how it works?

The GiveCash and ReceiveCash methods verify that the amount they're being asked to give or receive is valid. That way you can't ask a guy to receive a negative number, which would cause him to lose cash.

Compare the comments in this code to the class diagrams and illustrations of the Guy objects. If something doesn't make sense at first, take the time to really understand it.

There's an easier way to initialize objects with C#

Almost every object that you create needs to be initialized in some way. The Guy object is no exception—it's useless until you set its Name and Cash fields. It's so common to have to initialize fields that C# gives you a shortcut for doing that. It's called an **object initializer**, and the IDE's IntelliSense will help you add one.

Object initializers save you time and make your code more compact and easier to read...and the IDE helps you write them.

You're about to do an exercise where you create two Guy objects. You ***could*** use one `new` statement and two more statements to set its fields:

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

Instead, type this: `Guy joe = new Guy() {`

As soon as you add the left curly bracket, the IDE will pop up an IntelliSense window that shows all of the fields that you can initialize:

```
Guy joe = new Guy() { }
```

Choose the Cash field, set it to 50, and add a comma:

```
Guy joe = new Guy() { Cash = 50,
```

Now type a space—another IntelliSense window will pop up with the remaining field to set:

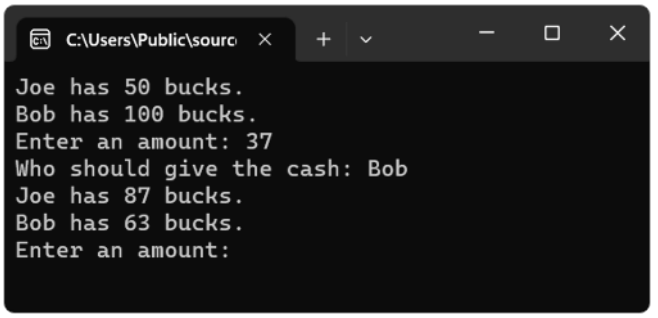
```
Guy joe = new Guy() { Cash = 50, }
```

Set the Name field and add the semicolon. You now have a single statement that initializes your object:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

This new declaration does the same thing as the three lines of code at the top of the page, but it's shorter and easier to read.

Now you have all of the pieces to build your console app that uses two instances of the Guy class. Here's what it will look like when it's running:



Here's how it works. It calls each Guy object's WriteMyInfo method. It reads an amount from the input and asks who to give the cash to, then calls one Guy object's GiveCash method, then the other Guy object's ReceiveCash method. It keeps going until the user enters a blank line, then it calls return to exit the app.

Exercise

This is Part 1 of a two-part exercise.

Here are the top-level statements for a console app that makes Guy objects give cash to each other.

Step 1: **Create a new console app** that *uses top-level statements*. Name it Guys.

Step 2: Add a new class to your app called Guy. Since your project is called Guys, your new class will be in the namespace Guys. Carefully **add all of the code from the Guy class** that we just showed you.

Step 3: Here's the code that goes into your app's *Program.cs* file. Carefully enter it, then **replace the `comments` in with code**—read each comment and write code that does exactly what it says. When you're done, you'll have a program that looks like the screenshot on the previous page.

```
// Create a new Guy object in a variable called joe
// Set its Name field to "Joe"
// Set its Cash field to 50

// Create a new Guy object in a variable called bob
// Set its Name field to "Bob"
// Set its Cash field to 100

while (true)
{
    // Call the WriteMyInfo methods for each Guy object

    Console.Write("Enter an amount: ");
    string? howMuch = Console.ReadLine();
    if (howMuch == "") return;
    // Use int.TryParse to try to convert the howMuch string? to an int
    // if it was successful (just like you did earlier in the chapter)
    {
        Console.Write("Who should give the cash: ");
        string? whichGuy = Console.ReadLine();
        if (whichGuy == "Joe")
        {
            // Call the joe object's GiveCash method and save the results
            // Call the bob object's ReceiveCash method with the saved results
        }
        else if (whichGuy == "Bob")
        {
            // Call the bob object's GiveCash method and save the results
            // Call the joe object's ReceiveCash method with the saved results
        }
        else
        {
            Console.WriteLine("Please enter 'Joe' or 'Bob'");
        }
    }
    else
    {
        Console.WriteLine("Please enter an amount (or a blank line to exit).");
    }
}
```

Replace all of the comments with code that does what the comments describe.

Add another `if` statement after an `else` to check for more than one condition. This first checks if whichGuy is Joe, then it checks if whichGuy is Bob, and if neither are true, it writes a line to the console.

Exercise Solution

Here are the top-level statements for your console app. It uses an infinite loop to keep asking the user how much cash to move between the Guy objects. If the user enters a blank line for an amount, the method executes a `return` statement, which causes Main to exit and the program to end.

```
using Guys;

(Guy joe = new Guy() ...
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
Guy bob = new Guy() { Cash = 100, Name = "Bob" };

while (true)
{
    joe.WriteMyInfo();
    bob.WriteMyInfo();
    Console.Write("Enter an amount: ");
    string? howMuch = Console.ReadLine();
    if (howMuch == "") return;
    if (int.TryParse(howMuch, out int amount))
    {
        Console.Write("Who should give the cash: ");
        string? whichGuy = Console.ReadLine();
        if (whichGuy == "Joe")
        {
            int cashGiven = joe.GiveCash(amount);
            bob.ReceiveCash(cashGiven);
        }
        else if (whichGuy == "Bob")
        {
            int cashGiven = bob.GiveCash(amount);
            joe.ReceiveCash(cashGiven);
        }
        else
        {
            Console.WriteLine("Please enter 'Joe' or 'Bob'");
        }
    }
    else
    {
        Console.WriteLine("Please enter an amount (or a blank line to exit).");
    }
}
```

When the app executes this return statement it ends the program, because console apps stop when the top-level statements finish running.

Here's the code where one Guy object gives cash from his pocket, and the other Guy object receives it.

Use the debugger to step through the code. Put breakpoints in the if, else if, and else blocks. Can you make it hit all three breakpoints?

Don't move on to the next part of the exercise until you have the first part working and you understand what's going on. It's worth taking a few minutes to use the debugger to step through the program and make sure you really get it.

Exercise

Here's the second part of the two-part exercise.

Now that you have your Guy class working, let's see if you can **reuse it in a betting game**. Look closely at this screenshot to see how it works and what it prints to the console.

```
Welcome to the casino. The odds are 0.75
The player has 100 bucks.
How much do you want to bet: 36
Bad luck, you lose.
The player has 64 bucks.
How much do you want to bet: 27
You win 54
The player has 91 bucks.
How much do you want to bet: 83
Bad luck, you lose.
The player has 8 bucks.
How much do you want to bet: 8
Bad luck, you lose.
The house always wins.
```

These are the odds to beat.

The player makes a double-or-nothing bet each round.

The program picks a random double from 0 to 1. If the number is greater than the odds, the player wins twice their bet, otherwise the player loses.

Create a new console app that uses top-level statements, then add the Guy class from your Guys project. Make sure you **add a using statement to the top of your *Program.cs* file** so you can use the Guy class.

In your *Program.cs*, declare two variables:

- A double variable called **`odds`** that stores the odds to beat set to .75
- A Guy variable called **`player`** for an instance of Guy named "The player" with 100 bucks.

Your app should write a line to the console welcoming the player and printing the odds. Then it should run this loop:

1. Call the Guy object's WriteMyInfo method to write the amount of cash the player has to the console.
2. Write a line to the console asking the player how much money to bet.
3. Read the line from the console into a string variable called `howMuch`.
4. Try to parse it into an int variable called `amount`.
5. If it parses, the player gives the amount to an int variable called `pot`. Only do steps 6 through 9 if `pot` is greater than zero.
6. Multiply `pot` by two, because it's a double-or-nothing bet.
7. Use Random.Shared to pick a random double value between 0 and 1.
8. If the random value is greater than `odds`, the player receives the pot.
9. If not, the player loses the amount they bet.
10. The program keeps running while the player has cash.

In step 5, you'll call the Guy object's *GiveCash method* to give the amount to bet. The GiveCash method won't give more cash than the guy has, so you don't need to check if the player has enough money. The Guy class will write a message to the console if it doesn't have enough cash to place the bet, so your app doesn't have to. Checking if the `pot` variable is greater than zero makes sure the bet is valid and the player has enough cash.

The loop ends when the player runs out of money, then the app prints a message: "The house always wins."

Exercise Solution

Here's the working code for the top-level statements in the betting game. Can you think of ways to make it more fun? See if you can figure out how to add additional players, or give different options for odds, or maybe you can think of something more clever. ***This is a chance to get creative!***

...and to get some practice. Getting practice writing code is the best way to become a great developer.

```
using Guys;

double odds = .75;
Random random = new Random();

Guy player = new Guy() { Cash = 100, Name = "The player" };

Console.WriteLine("Welcome to the casino. The odds are " + odds);
while (player.Cash > 0)
{
    player.WriteMyInfo();
    Console.Write("How much do you want to bet: ");
    string? howMuch = Console.ReadLine();
    if (int.TryParse(howMuch, out int amount))
    {
        int pot = player.GiveCash(amount) * 2;
        if (pot > 0)
        {
            if (Random.Shared.NextDouble() > odds)
            {
                int winnings = pot;
                Console.WriteLine("You win " + winnings);
                player.ReceiveCash(winnings);
            } else
            {
                Console.WriteLine("Bad luck, you lose.");
            }
        }
    } else
    {
        Console.WriteLine("Please enter a valid number.");
    }

}
Console.WriteLine("The house always wins.");
```

Was your code a little different than ours? If it still works and produces the right output, that's OK! There are many different ways to write the same program.

...and as you get further along in the book and the exercise solutions get longer, your code will look more and more different from ours. Remember, it's always OK to look at our solution when you're working on an exercise!

Brain Power

Is Guy really the best name for the class? Why or why not? Can you think of a better name for it?

Sharpen your pencil

Here's an app that writes three lines to the console. Your job is to figure out what it writes, without using a computer. Start at the first line of the Main method and keep track of the values of each of the fields in the objects as it runs.

```
Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
foxtrot.Bamboo(foxtrot.Zippo);

Pizzazz november = new Pizzazz() { Zippo = 3 };
Abracadabra tango = new Abracadabra() { Vavavoom = 4 };

while (tango.Lala(november.Zippo))
{
    november.Zippo *= -1;
    november.Bamboo(tango.Vavavoom);
    foxtrot.Bamboo(november.Zippo);
    tango.Vavavoom -= foxtrot.Zippo;
}

Console.WriteLine("november.Zippo = " + november.Zippo);
Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
class Pizzazz
{
    public int Zippo;

    public void Bamboo(int eek)
    {
        Zippo += eek;
    }
}

class Abracadabra
{
    public int Vavavoom;

    public bool Lala(int floq)
    {
        if (floq < Vavavoom)
        {
            Vavavoom += floq;
            return true;
        }
        return false;
    }
}
```

What does this program write to the console?

november.Zippo = ...

foxtrot.Zippo = ...

tango.Vavavoom = ...

To find the solution, enter the program into Visual Studio and run it. If you didn't get the answer right, step through the code line by line and add watches for each of the object's fields.

If you don't want to type the whole thing in, you can download it from GitHub: *https://github.com/head-first-csharp/fifth-edition*

Try this a few times with each chatbot. Does it generate the same code each time?

Sens-AI

Ask an AI chatbot to do the exercise to write the betting game code for you

AI chatbots like Copilot, ChatGPT, and Gemini can do more than answer questions. They can draw pictures, search for information, and even **generate code for you**. Let's use the betting game to explore how this works.

Step 1: Download the PDF of the first four chapters of this book.

We made the first four chapters of this book available as a free PDF. Download it from our GitHub page:

https://github.com/head-first-csharp/fifth-edition

Step 2: Use the complete text of Part 1 of the exercise as a prompt.

Open the PDF and find Part 1 of the two-part exercise in this chapter. Part 1 starts with this sentence:

Here are the top-level statements for a console app that makes Guy objects give cash to each other.

Select all of the text for Part 1—including all of the code—starting with "*Here are the top-level statements*" and ending with the closing bracket `}` at the bottom.

Open Copilot (*https://copilot.microsoft.com*), ChatGPT (*https://chat.openai.com*), Gemini (*https://gemini.google.com*), or any other AI chatbot. Paste in all of the text that you copied as a prompt. The chatbot should generate code for a Guy class, and code for the top-level statements that use it.

Step 3: Create a new Console App project and paste the generated code into it.

Create a new project to test out the code that the AI chatbot generated for you. Create a Guy class in your project, then copy the code that the AI generated for its Guy class and paste it in. Finally, copy the code that the AI generated for its top-level statements into your *Program.cs*.

Take the time to understand the code that the AI generated. Try this a few times and compare the results. That's a really good way to learn.

Step 4: Use the AI safely—*trust but verify*.

Before you run your app, look at the code. Does it look right?

When we did this ourselves, all three AI chatbots generated working code that did exactly what we asked it to do. However, one of the chatbots ignored our instruction to generate top-level statements and created a console app with a Main method. The app still worked, though!

Also, two of the chatbots used object initializers, while one of them set the fields directly. Our instructions didn't tell it exactly what to do, and either of those choices will work just fine.

Step 5: Finish the betting game.

Find Part 2 of the exercise in the PDF, copy all of the text, and paste it into the chatbot ***as a follow-up prompt*** (so don't start a new chat). Copy the code that it generates and paste it into your *Program.cs* file. Before you run your app, **carefully read all of the code**. Does it look like it works? Did the AI do things differently than you did?

When we pasted our Part 2 prompt, the betting game worked perfectly. All three chatbots were even smart enough to ignore the instruction in Part 2 to look closely at the screenshot to see how the game works and what it prints to the console, even though we didn't actually include the contents of the screenshot in our prompt.

Did the AI generate code that behaves the same way as the code you wrote for the exercise, but is structured differently? Compare its code to yours—that might give you ideas about how you can *refactor* your code.

If an AI chatbot is smart enough to write code for me, why should I ***bother writing code****? Why should anyone even learn to code in the first place?*

It's often *easier to write the code* that you want than it is to engineer a prompt to generate that code.

Creating a prompt isn't always easy. In fact, there's an entire growing field of study called **prompt engineering** that's all about designing prompts to get an AI to do what you want it to do.

Creating a great prompt can include many steps. You need to test the specific language that you're using, examine the output, then refine it to get the AI to give you the output that you're looking for. Prompt engineering can be a difficult and often tedious process, because a very small change in a prompt can make a huge difference in the output that the AI generates.

We know this because ***every coding exercise in this book is a prompt***. The only difference is that instead of building those prompts for an artificial intelligence, we built them to be used by a real, smart, *human* intelligence—namely, *you*!

Every single exercise in this book went through a painstaking process to make sure you have all of the information that you need to do it. When we're building an exercise, we start by writing the code for the solution. Believe it or not, that's the quickest part of creating the exercise. Then we go through a process of creating instructions. We usually have to go through many versions of each exercise before we end up with the version that you see. It's a time-consuming and painstaking process—and sometimes we don't get it right. We got feedback from readers of the first four editions of this book (especially the early editions) who had trouble doing exercises because our instructions were unclear, difficult to follow, and on a few occasions, even missed some steps.

In this edition, we tested our exercises by pasting them into different AI chatbots to make sure we didn't accidentally leave out any instructions. If the AI can generate code for a correct solution, it means we included everything you need to do it too.

In every single case, writing the code for each exercise was *much* easier than the prompt engineering that we had to do to craft the finished exercise.

Exercise

Decide for yourself if prompt engineering is harder than writing code. We gave you this method earlier in the chapter:

```
int Multiply(int factor1, int factor2)
{
   int product = factor1 * factor2;
   return product;
}
```

On our first try, we had to ask four follow-up questions before we could get ChatGPT to give us exactly this output.

Start a new session with an AI chatbot and give it a prompt to create exactly this method. Make sure it doesn't include comments or any additional code. Ask it follow-up questions until it gives you exactly this output.

Repeat the same exercise with the loop from the Sens-AI element in Chapter 2. Can you get the AI to generate it?

Use the C# Interactive window or csi to run C# code

If you're running Windows, open Developer Command Prompt or Developer PowerShell from the Start menu to make sure csi is in your path.

If you just want to run some C# code, you don't always need to create a new project in Visual Studio. Any C# code entered into the **C# Interactive window** is run immediately. You can open it by choosing View >> Other Windows >> C# Interactive. Try it now, and **paste in the code** from the exercise solution to see the output. You can call methods and enter other statements too.

VSCode doesn't have a C# Interactive window, but you can run `csi` from Terminal to use the `dotnet` C# interactive compiler.

C# Interactive

```
.     public bool Lala(int floq)
.     {
.         if (floq < Vavavoom)
.         {
.             Vavavoom += floq;
.             return true;
.         }
.         return false;
.     }
. }
november.Zippo = 4
foxtrot.Zippo = 8
tango.Vavavoom = -1
> |
```

110 %

C# Interactive | Find Results 1 | Error List

```
andrew — mono --gc-params=nursery-size=64m /Library/Frameworks/Mono.framework/Versions/6.12.0/lib/mono/4.5/
Andrews-MBP ~ % csi
Microsoft (R) Visual C# Interactive Compiler version 3.9.0-6.
Copyright (C) Microsoft Corporation. All rights reserved.

Type "#help" for more information.
> class Pizzazz
. . . . . .
> class Abracadabra
. . . . . .
> Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
> > > > > > . . . . . .
> Console.WriteLine("november.Zippo = " + november.Zippo);
november.Zippo = 4
> Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
foxtrot.Zippo = 8
> Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
tango.Vavavoom = -1
>
```

Paste in each class. You'll see periods for each pasted line.

Only the first pasted line is printed, so we pasted each Console.WriteLine statement separately to see the output.

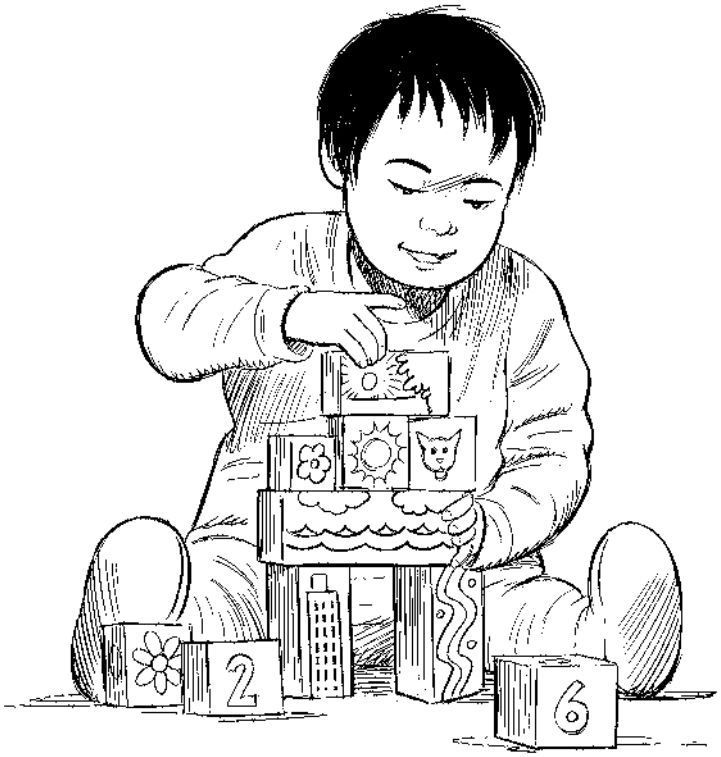
You can also run an interactive C# session from the command line. On Windows, search the Start menu for `Developer Command Prompt`, start it, and then type **`csi`**. On macOS or Linux, run **`csi`** from the Terminal. You can paste the Pizzazz, Abracadabra, and Program classes from the previous exercise directly into the prompt, then paste in the code that you want to run.

Bullet Points

- Use the **`new` keyword** to create instances of a class. A program can have many instances of the same class.
- Each **instance** has all of the methods from the class and gets its own copies of each of the fields.
- When you included `new Random();` in your code, you were creating an **instance of the Random class**.
- Use the **`static` keyword** to declare a field or method in a class as static. You don't need an instance of that class to access static methods or fields.
- When a field is **static**, there's only one copy of it shared by all instances. When you include the `static` keyword in a class declaration, all of its members must be static.
- Fields and methods of a class are called its **members**.
- If you remove the `static` keyword from a static field, it becomes an **instance field**.
- When your program creates an object, it lives in a part of the computer's memory called the **heap**.
- Visual Studio helps you add **XML documentation** (or **XMLDoc**) to your fields and methods, and displays it in its IntelliSense window.
- **Class diagrams** help you plan out your classes and make them easier to work with.
- When you change the structure of your code without altering its behavior, it's called **refactoring**. Advanced developers constantly refactor their code.
- **Object initializers** save you time and make your code more compact and easier to read.

4 data, types, objects, and references

Managing your app's data



Data and objects are the building blocks of your apps.

What would your apps be without data? Think about it for a minute. Without data, your programs are…well, it's actually hard to imagine writing code without data. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types** and **references**, see how to work with data in your program, and even learn a few more things about **objects** (guess what…objects are data too!).

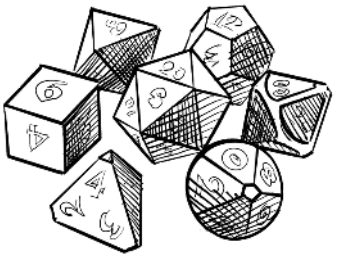
Owen could use our help!

Owen is a game master—a really good one. He hosts a group that meets at his place every week to play different **role-playing games** (or **RPGs**), and like any good game master, he works hard to keep things interesting for the players.

Storytelling, fantasy, and mechanics

Owen is a particularly good storyteller. Over the last few months, he's created an intricate fantasy world for his party, but he's not so happy with the mechanics of the game that they've been playing.

Can we find a way to help Owen improve his RPG?



Ability score (like strength, stamina, charisma, and intelligence) is an important mechanic in a lot of role-playing games. Players frequently roll dice and use a formula to determine their character's scores.

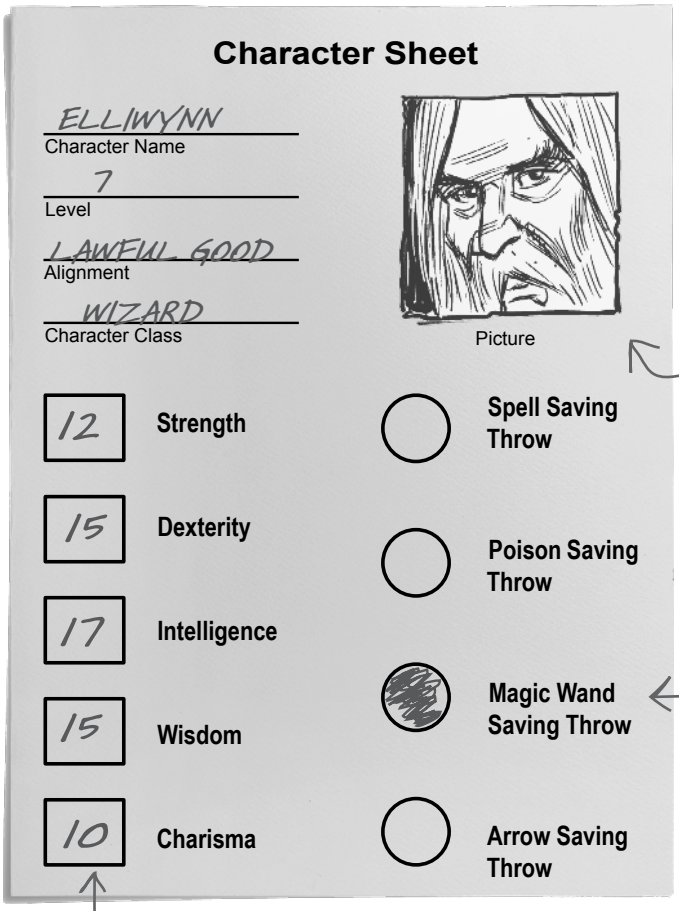
Character sheets store different types of data on paper

If you've ever played an RPG, you've seen character sheets: a page with details, statistics, background information, and any other notes you might see about a character. If you wanted to make a class to hold a character sheet, what types would you use for the fields?

CharacterSheet

CharacterName
Level
PictureFilename
Alignment
CharacterClass
Strength
Dexterity
Intelligence
Wisdom
Charisma
SpellSavingThrow
PoisonSavingThrow
MagicWandSavingThrow
ArrowSavingThrow

ClearSheet
GenerateRandomScores



This box is for a picture of the character. If you were building a C# class for a character sheet, you could save that picture in an image file.

In the RPG that Owen plays, saving throws give players a chance to roll dice and avoid certain types of attacks. This character has a magic wand saving throw, so the player filled in this circle.

Players create characters by rolling dice for each of their ability scores, which they write in these boxes.

Brain Power

Look at the fields in the CharacterSheet class diagram. What type would you use for each field?

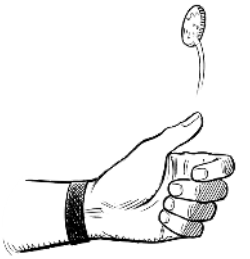
A variable's type determines what kind of data it can store

There are many **types** built into C#, and you'll use them to store many different kinds of data. You've already seen some of the most common ones, like int, string, bool, and float. There are a few others that you haven't seen, and they can really come in handy too.

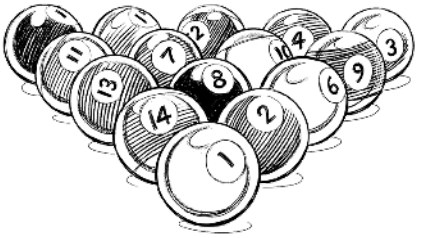
Here are some types you'll use a lot.

Better a witty fool, than a foolish wit.

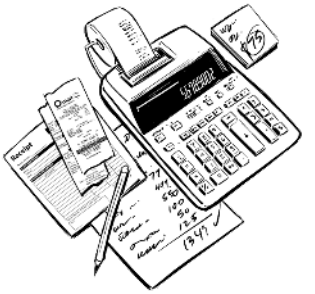
- **string** can hold text of any length (including the empty string `""`).



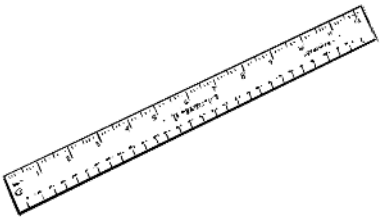
- **bool** is a Boolean value—it's either true or false. You'll use it to represent anything that only has two options: it can either be one thing or another, but nothing else.



- **int** can store any **integer** from −2,147,483,648 to 2,147,483,647. Integers don't have decimal points.



- **double** can store **real** numbers from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with up to 16 significant digits. It's a really common type when you're working with XAML properties.



- **float** can store **real** numbers from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with up to 8 significant digits.

Brain Power

Why do you think C# has more than one type for storing numbers that have a decimal point?

C# has several types for storing integers

C# has several different types for integers, as well as int. This may seem a little odd (pun intended). Why have so many types for numbers without decimals? For most of the programs in this book, it won't matter if you use an int or a long. If you're writing a program that has to keep track of millions and millions of integer values, then choosing a smaller integer type like byte instead of a bigger type like long can save you a lot of memory.

- ***byte*** can store any **integer** between 0 and 255.
- ***sbyte*** can store any **integer** from –128 to 127.
- ***short*** can store any **integer** from –32,768 to 32,767.
- ***long*** can store any **integer** from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Notice how we're saying "integer" and not "whole number"? We're trying to be really careful—our high school math teachers always told us that integers are any numbers that can be written without a fraction, while whole numbers are integers starting at 0, and do not include negative numbers.

byte only stores small whole numbers from 0 to 255.

If you need to store a larger number, you can use a short, which stores integers from –32,768 to 32,767.

Long also stores integers, but it can store huge values.

Did you notice that byte only stores positive numbers, while sbyte stores negative numbers? They both have 256 possible values. The difference is that, like short and long, sbyte can have a negative sign—which is why those are called **signed** types, (the "s" in sbyte stands for signed). Just like byte is the **unsigned** version of sbyte, there are unsigned versions of short, int, and long that start with "u":

- ***ushort*** can store any **whole number** from 0 to 65,535.
- ***uint*** can store any **whole number** from 0 to 4,294,967,295.
- ***ulong*** can store any **whole number** from 0 to 18,446,744,073,709,551,615.

Types for storing really HUGE and really tiny numbers

Sometimes float just isn't precise enough. Believe it or not, sometimes 10^{38} isn't big enough and 10^{-45} isn't small enough. A lot of programs written for finance or scientific research run into these problems all the time, so C# gives us different **floating-point types** to handle huge and tiny values:

- ***float*** can store any number from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 6–9 significant digits.
- ***double*** can store any number from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with 15–17 significant digits.
- ***decimal*** can store any number from $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ with 28–29 significant digits. When your program **needs to deal with money or currency**, you always want to use a decimal to store the number.

The decimal type has a lot more precision (way more significant digits), which is why it's appropriate for financial calculations.

Floating-Point Numbers Up Close

The float and double types are called "floating-point" because the decimal point can move (as opposed to a "fixed-point" number, which always has the same number of decimal places). That—and, in fact, a lot of stuff that has to do with floating-point numbers, especially precision—may seem a little ***weird***, so let's dig into the explanation.

"Significant digits" represents the precision of the number: 1,048,415, 104.8415, and .0000001048415 all have seven significant digits. So when we say a float can store real numbers as big as 3.4×10^{38} or as small as -1.5×10^{-45}, that means it can store numbers as big as eight digits followed by 30 zeros, or as small as 37 zeros followed by eight digits.

If it's been a while since you've used exponents, 3.4×10^{38} means 34 followed by 37 zeros, and -1.5×10^{-45} is -.00...(40 more zeros)...0015.

The float and double types can also have special values, including both positive and negative zero, positive and negative infinity, and a special value called **NaN (not-a-number)** that represents, well, a value that isn't a number at all. They also have static methods that let you test for those special values. Try running this loop:

```
for (float f = 10; !float.IsInfinity(f); f *= f)
{
    Console.WriteLine(f);
}
```

Now try that same loop with double:

```
for (double d = 10; !double.IsInfinity(d); d *= d)
{
    Console.WriteLine(d);
}
```

Let's talk about strings

You've written code that works with **strings**. So what, exactly, is a string?

In any .NET app, a string is an object. Its full class name is System.String—in other words, the class name is String and it's in the System namespace (just like the Random class you used earlier). When you use the C# `string` keyword, you're working with System.String objects. In fact, you can replace `string` with `System.String` in any of the code you've written so far and it will still work! (The `string` keyword is called an *alias*—as far as your C# code is concerned, `string` and `System.String` mean the same thing.)

There are also two special values for strings: an empty string, `""` (or a string with no characters), and a null string, or a string that isn't set to anything at all. We'll talk more about null later in the chapter.

Strings are made up of characters—specifically, Unicode characters (which you'll learn a lot more about later in the book). Sometimes you need to store a single character like `Q` or `j` or `$`, and when you do you'll use the **char** type. Literal values for char are always inside single quotes (`'x'`, `'3'`). You can include **escape sequences** in the quotes too (`'\n'` is a line break, `'\t'` is a tab). You can write an escape sequence in your C# code using two characters, but your program stores each escape sequence as a single character in memory.

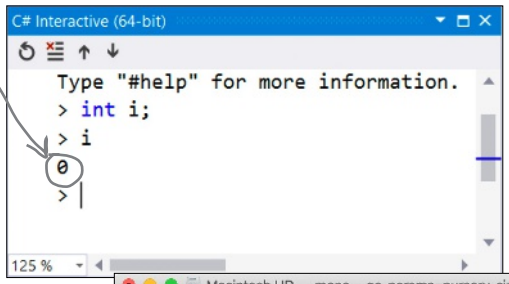
And finally, there's one more important type: **object**. If a variable has object as its type, ***you can assign any value to it***. The `object` keyword is also an alias—it's the same as `System.Object`.

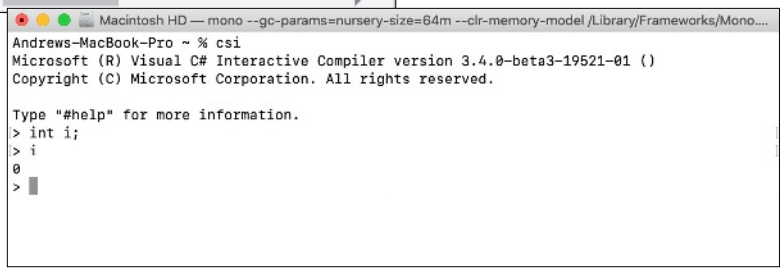
Sharpen your pencil

Sometimes you declare a variable and set its value in a single statement like this: `int i = 37;`—but you already know that you don't have to set a value. What happens if you use the variable without assigning a value? Let's find out! Use the **C# Interactive window** (or the **csi** if you're using VSCode) to declare a variable and check its value.

We wrote in the first answer for you.

```
0 ..... int i;
..... long l;
..... float f;
..... double d;
..... decimal m;
..... byte b;
..... char c;
..... string s;
..... bool t;
```





Start the C# Interactive window (from the View >> Other Windows menu) or run `csi` from the command line. Declare each variable, then enter the variable name to see its default value. Write the default value for each type in the space provided.

A literal is a value written directly into your code

A **literal** is a number, string, or other fixed value that you include in your code. You've already used plenty of literals—here are some examples of numbers, strings, and other literals that you've used:

```
int number = 15;
string result = "the answer";
public bool GameOver = false;
Console.Write("Enter the number of cards to pick: ");
if (value == 1) return "Ace";
```

Can you spot all of the literals in these statements from code you've written in previous chapters? The last statement has two literals.

So when you type `int i = 5;`, the **5** is a literal.

Use suffixes to give your literals types

Go back to the first loop you wrote in the "Up Close" section and change 10 to 10***D***:

```
for (float f = 10D; float.IsFinite(f); f *= f)
```

Now your *code* ***will have a syntax error*** *and won't build*. The C# compiler error mentions a "literal of type double." That's because **literals have types**. Every literal is automatically assigned a type, and C# has rules about how you can combine different types. You can see for yourself how that works. Add this line to any C# program:

```
int wholeNumber = 14.7;
```

When you try to build your program, the IDE will show you this error in the Error List:

CS0266	Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

The IDE is telling you is that the literal 14.7 has a type—it's a double. You can use a suffix to change its type—try changing it to a float by sticking an F on the end (14.7F) or a decimal by adding M (14.7M—the M actually stands for "money"). The error message now says it can't convert float or decimal.

C# assumes that an integer literal without a suffix (like 371) is an int, and one with a decimal point (like 27.4) is a double.

Sharpen your pencil Solution

```
0 ..... int i;
0 ..... long l;
0 ..... float f;
0 ..... double d;
0 ..... decimal m;
0 ..... byte b;
'\0' ..... char c;
null ..... string s;
false ..... bool t;
```

If you used the C# command line on Mac or Linux, you might see '\x0' instead of '\0' as the default value for char. We'll take a deep dive into exactly what this means later in the book when we talk about Unicode.

Sharpen your pencil

C# has dozens of **reserved words called keywords**. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

`namespace`	
`for`	
`class`	
`else`	
`new`	
`using`	
`if`	
`while`	

If you really want to use a reserved keyword as a variable name, put @ in front of it, but that's as close as the compiler will let you get to the reserved word. You can also do that with nonreserved names, if you want to.

Sharpen your pencil Solution

C# has dozens of **reserved words called keywords**. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

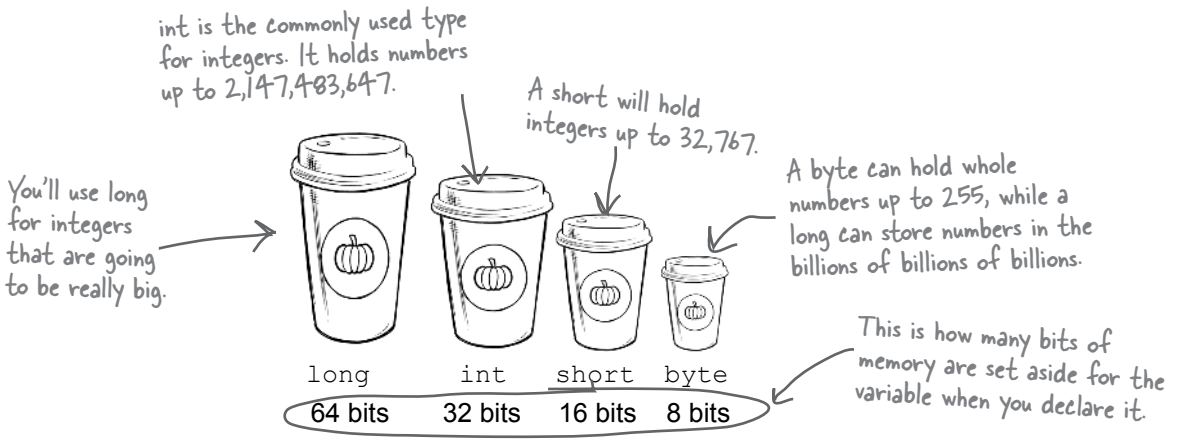
namespace	All of the classes and methods in a program are inside a namespace. Namespaces help make sure that the names you are using in your program don't clash with the ones in the .NET Framework or other classes.
for	This lets you do a loop that executes three statements. First it declares the variable it's going to use, then there's the statement that evaluates the variable against a condition. The third statement does something to the value.
class	Classes contain methods and fields, and you use them to instantiate objects. Fields are what objects know and methods are what they do.
else	A block of code that starts with else must immediately follow an if block, and will get executed if the if statement preceding it fails.
new	You use this to create a new instance of an object.
using	This is a way of listing off all of the namespaces you are using in your program. A using statement lets you use classes from various parts of the .NET Framework.
if	This is one way of setting up a conditional statement in a program. It says if one thing is true, do one thing; if not, do something else.
while	while loops are loops that keep on going as long as the condition at the beginning of the loop is true.

A variable is like a data to-go cup

All of your data takes up **space in memory**. (Remember the heap from the previous chapter?) So part of your job is to think about how *much* space you're going to need whenever you use a string or a number in your program. That's one of the reasons you use variables. They let you set aside enough space in memory to store your data.

Not all data ends up on the heap. Value types usually keep their data in another part of memory called the stack. You'll learn all about that later in the book.

Think of a variable like a cup that you keep your data in. C# uses a bunch of different kinds of cups to hold different kinds of data. Just like the different sizes of cups at a coffee shop, there are different sizes of variables too.



Convert this!

Use the Convert class to explore bits and bytes

You've always heard that programming is about 1s and 0s. .NET has a **static Convert class** that converts between different numeric data types. Let's use it to see an example of how bits and bytes work. Type these Convert method calls into the Visual Studio C# Interactive window or CSI.

A bit is a single 1 or 0. A byte is 8 bits, so a byte variable holds an 8-bit number, which means it's a number that can be represented with up to 8 bits. What does that look like? Let's use the Convert class to convert some binary numbers to bytes:

```
Convert.ToByte("10111", 2)    // returns 23
Convert.ToByte("11111111", 2) // returns 255
```

The first argument to Convert.ToByte is the number to convert, and the second is its base. Binary numbers are base 2.

Bytes can hold numbers between 0 and 255 because they use 8 bits of memory—an 8-bit number is a binary number between 0 and 11111111 binary (or 0 and 255 decimal).

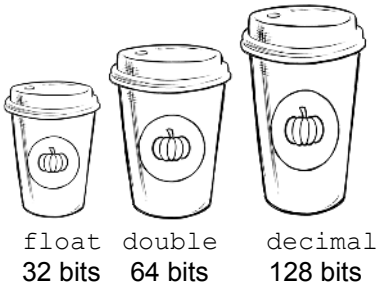
A short is a 16-bit value. Let's use `Convert.ToInt16` to convert the binary value 111111111111111 (15 1s) to a short. An int is a 32-bit value, so we'll use `Convert.ToInt32` to convert the 31 1s to an int:

```
Convert.ToInt16("111111111111111", 2)                // returns 32767
Convert.ToInt32("1111111111111111111111111111111", 2) // returns 2147483647
```

Other types come in different sizes too

Numbers that have decimal places are stored differently than integers, and the different floating-point types take up different amounts of memory. You can handle most of your numbers that have decimal places using **float**, the smallest data type that stores decimals. If you need to be more precise, use a **double**. If you're writing a financial application where you'll be storing currency values, you'll always want to use the **decimal** type.

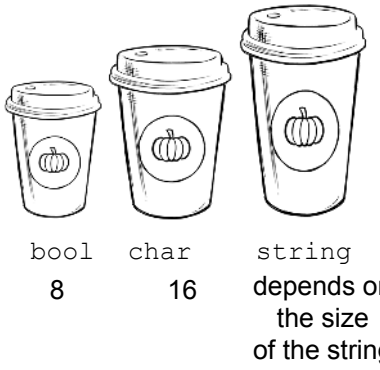
Oh, and one more thing: ***don't use double for money or currency, only use decimal.***



These types are for fractions. They can also be used to store very large numbers. Larger variables store more decimal places.

We've talked about strings, so you know that the C# compiler also can handle **characters and non-numeric types**. The char type holds one character, and string is used for lots of characters "strung" together. There's no set size for a string object—it expands to hold as much data as you need to store in it. The bool data type is used to store true or false values, like the ones you've used for your `if` statements.

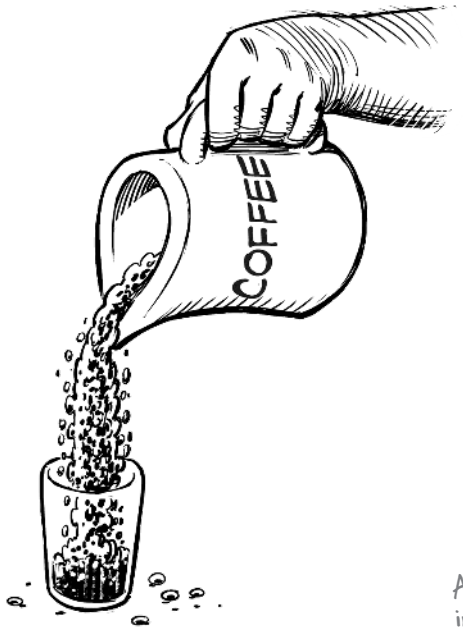
C# also has types for storing data that is not numeric.



Strings can be big...REALLY big! C# uses a 32-bit integer to keep track of the string length, so the maximum string length is 2,147,483,647 characters.

The different floating-point types take up different amounts of memory: float is smallest, and decimal is largest.

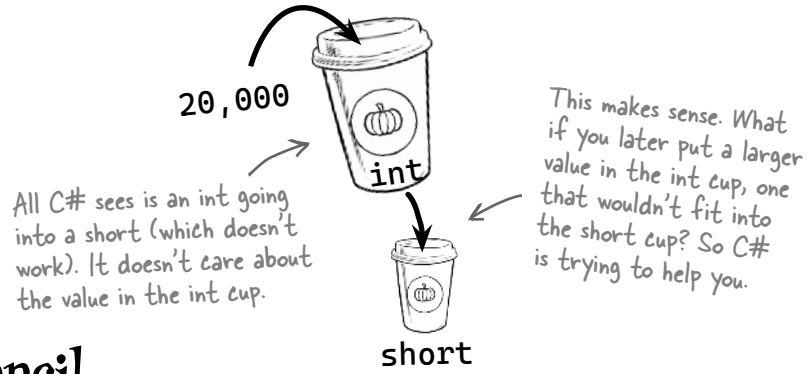
10 pounds of data in a 5-pound bag



When you declare your variable as one type, the C# compiler **allocates** (or reserves) all of the memory it would need to store the maximum value of that type. Even if the value is nowhere near the upper boundary of the type you've declared, the compiler will see the cup it's in, not the number inside. So this won't work:

```
int leaguesUnderTheSea = 20000;
short smallerLeagues = leaguesUnderTheSea;
```

20,000 would fit into a **short**, no problem. But because **leaguesUnderTheSea** is declared as an int, C# sees it as int-sized and considers it too big to put in a short container. The compiler won't make those translations for you on the fly. You need to make sure that you're using the right type for the data you're working with.



Sharpen your pencil

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong.

```
int hours = 24;

short y = 78000;

bool isDone = yes;

short RPM = 33;

int balance = 345667 - 567;
```

```
string taunt = "your mother";

byte days = 365;

int radius = 3;

char initial = 'S';

string months = "12";
```

Casting lets you copy values that C# can't automatically convert to another type

Let's see what happens when you try to assign a decimal value to an int variable.

Do this!

1. Create a new Console App project and add this code to your *Program.cs*:

```
float myFloatValue = 10;
int myIntValue = myFloatValue;
Console.WriteLine("myIntValue is " + myIntValue);
```

Implicit conversion means C# has a way to automatically convert a value to another type without losing information.

2. Try building your program. You should get the same CS0266 error you saw earlier:

```
CS0266  Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)
```

Look closely at the last few words of the error message: "are you missing a cast?" That's the C# compiler giving you a really useful hint about how to fix the problem.

3. Make the error go away by **casting** the decimal to an int. You do this by adding the type that you want to convert to in parentheses: **`(int)`**. Once you change the second line so it looks like this, your program will compile and run:

```
int myIntValue = (int) myFloatValue;
```

Here's where you cast the decimal value to an int.

When you cast a floating-point value to an int, it *rounds the value down* to the nearest integer.

So what happened?

The C# compiler won't let you assign a value to a variable if it's the wrong type—even if that variable can hold the value just fine! It turns out that a LOT of bugs are caused by type problems, and **the compiler is helping** by nudging you in the right direction. When you use casting, you're essentially saying to the compiler that you know the types are different, and promising that in this particular instance it's OK for C# to cram the data into the new variable.

Sharpen your pencil Solution

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong.

```
short y = 78000;
```

The short type holds numbers from –32,767 to 32,768. This number's too big!

```
byte days = 365;
```

A byte can only hold a value between 0 and 255. You'll need a short for this.

```
bool isDone = yes;
```

You can only assign a value of "true" or "false" to a bool.

When you cast a value that's too big, C# adjusts it to fit its new container

You've already seen that a float can be cast to an int. It turns out that *any* number can be cast to *any other* number. That doesn't mean the ***value*** stays intact through the casting, though. Say you have an int variable set to 365. If you cast it to a byte variable (max value 255), instead of giving you an error, the value will just **wrap around**. 256 cast to a byte will have a value of 0, 257 will be converted to 1, 258 to 2, etc., up to 365, which will end up being **109**. Once you get back to 255 again, the conversion value "wraps" back to zero.

If you use + (or *, /, or -) with two different numeric types, the operator **automatically converts** the smaller type to the bigger one. Here's an example:

```
int myInt = 36;
float myFloat = 16.4F;
myFloat = myInt + myFloat;
```

Since an int can fit into a float but a float can't fit into an int, the + operator converts `myInt` to a float before adding it to `myFloat`.

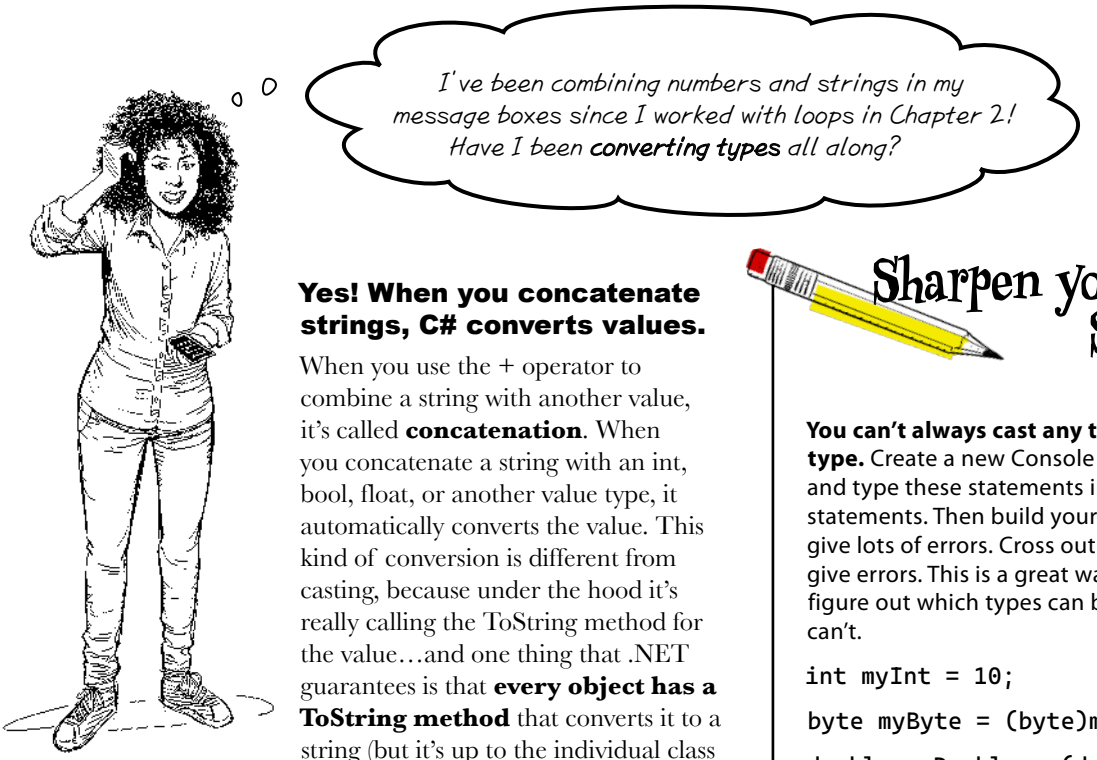
Sharpen your pencil

You can't always cast any type to any other type.

Create a new Console App project and type these statements into its top-level statements. Then build your program—it will give lots of errors. Cross out the ones that give errors. This is a great way to help you figure out which types can be cast, and which can't.

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)myLong;
myString = myString + myInt + myByte +
myDouble + myChar;
```

You can read a lot more about the different C# value types here—it's worth taking a look:
https://docs.microsoft.com/dotnet/csharp/language-reference/keywords/value-types



Yes! When you concatenate strings, C# converts values.

When you use the + operator to combine a string with another value, it's called **concatenation**. When you concatenate a string with an int, bool, float, or another value type, it automatically converts the value. This kind of conversion is different from casting, because under the hood it's really calling the ToString method for the value…and one thing that .NET guarantees is that **every object has a ToString method** that converts it to a string (but it's up to the individual class to determine if that string makes sense).

Wrap it yourself!

There's no mystery to how casting "wraps" the numbers—you can do it yourself. Just open up any calculator app that has a Mod button (which does a modulus calculation—sometimes in a Scientific mode), and calculate 365 Mod 256.

Sharpen your pencil Solution

You can't always cast any type to any other type. Create a new Console App project and type these statements into its top-level statements. Then build your program—it will give lots of errors. Cross out the ones that give errors. This is a great way to help you figure out which types can be cast, and which can't.

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)
myLong;
myString = myString + myInt +
myByte + myDouble + myChar;
```

C# does some conversions automatically

There are two important conversions that don't require you to do casting. The first is the automatic conversion that happens any time you use arithmetic operators, like in this example:

```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
Console.WriteLine("The answer is " + d);
```

The – operator subtracted the short from the long, and the = operator converted the result to a double.

The other way C# converts types for you automatically is when you use the + operator to **concatenate** strings (which just means sticking one string on the end of another, like you've been doing with message boxes). When you use + to concatenate a string with something that's another type, it automatically converts the numbers to strings for you. Here's an example—try adding these lines to any C# program. The first two lines are fine, but the third one won't compile:

```
long number = 139401930;
string text = "Player score: " + number;
text = number;
```

The C# compiler gives you this error on the third line:

CS0029 Cannot implicitly convert type 'long' to 'string'

ScoreText.text is a string field, so when you used the + operator to concatenate a string it assigned the value just fine. But when you try to assign `x` to it directly, it doesn't have a way to automatically convert the long value to a string. You can convert it to a string by calling its ToString method.

there are no Dumb Questions

Q: You used the Convert.ToByte, Convert.ToInt32, and Convert.ToInt64 methods to convert strings with binary numbers into integer values. Can you convert integer values back to binary?

A: Yes. The Convert class has a **Convert.ToString method** that converts many different types of values to strings. The IntelliSense pop-up shows you how it works:

```
Console.WriteLine(Convert.ToString(8675309, 2));
```

▲ 26 of 36 ▼ string Convert.ToString(int value, **int toBase**)
Converts the value of a 32-bit signed integer to its equivalent string representation in a specified base.
toBase: *The base of the return value, which must be 2, 8, 10, or 16.*

So `Convert.ToString(255, 2)` returns the string "11111111," and `Convert.ToString(8675309, 2)` returns the string "100001000101111111101101"—try experimenting with it to get a feel for how binary numbers work.

When you call a method, the arguments need to be compatible with the types of the parameters

In Chapter 3, you used the Random class to choose a random number from 1 up to (but not including) 5, which you used to pick a suit for a playing card:

```
int value = Random.Shared.Next(1, 5);
```

Try changing the first argument from `1` to `1.0`:

```
int value = Random.Shared.Next(1.0, 5);
```

You're passing a double literal to a method that's expecting an int value. So it shouldn't surprise you that the compiler won't build your program—instead, it shows an error:

```
CS1503   Argument 1: cannot convert from 'double' to 'int'
```

Sometimes C# can do the conversion automatically. It doesn't know how to convert a double to an int (like converting 1.0 to 1), but it does know how to convert an int to a double (like converting 1 to 1.0). More specifically:

- The C# compiler knows how convert an integer to a floating-point type.
- And it knows how to convert an integer type to another integer type, or a floating-point type to another floating-point type.
- But it can only do those conversions if the type it's converting from is the same size as or smaller than the type it's converting to. So, it can convert an int to a long or a float to a double, but it can't convert a long to an int or a double to a float.

But Random.Shared.Next isn't the only method that will give you compiler errors if you try to pass it a variable whose type doesn't match the parameter. *All* methods will do that, ***even the ones you write yourself***. Add this method to a console app's top-level statements:

```
int MyMethod(bool add3) {
   int value = 12;

   if (add3)
      value += 3;
   else
      value -= 2;

   return value;
}
```

Try passing it a string or long—you'll get one of those CS1503 errors telling you it can't convert the argument to a bool.

Some folks have trouble remembering **the difference between a parameter and an argument**. So just to be clear:

A *parameter* is what you define in your method. An *argument* is what you pass to it. You can pass a byte argument to a method with an int parameter.

When the compiler gives you an "invalid argument" error, it means that you tried to call a method with variables whose types didn't match the method's parameters.

there are no Dumb Questions

Q: That last `if` statement only said `if (add3)`. Is that the same thing as `if (add3 == true)`?

A: Yes. Let's take another look at that `if/else` statement:

```
if (add3)
   value += 3;
else
   value -= 2;
```

An `if` statement always checks if something's true. So because the type of the `add3` variable is bool, it evaluates to either true or false, which means we didn't have to explicitly include `== true`.

You can also check if something's false using ! (an exclamation point, or the NOT operator). Writing `if (!add3)` is the same thing as writing `if (add3 == false)`.

In our code examples from now on, if we're using the conditional test to check a Boolean variable, you'll usually just see us write `if (add3)` or `if (!add3)`, and not use == to explicitly check to see if the Boolean is true or false.

Q: You didn't include curly braces in the `if` or `else` blocks either. Does that mean they're optional?

A: Yes—but only if there's a single statement in the `if` or `else` block. We could leave out the `{` curly braces `}` because there was just one statement in the `if` block (`return 45;`) and one statement in the `else` block (`return 61;`). If we wanted to add another statement to one of those blocks, we'd have to use curly braces for it:

```
if (add3)
   value += 3;
else {
   Console.WriteLine("Subtracting 2");
   value -= 2;
}
```

Be careful when you leave out curly braces because it's easy to accidentally write code that doesn't do what you want it to do. It never hurts to add curly braces, but it's also good to get used to seeing `if` statements both with and without them.

Bullet Points

- There are value types for variables that hold different sizes of numbers. The biggest numbers should be of type long and the smallest ones (up to 255) can be declared as bytes.
- Every value type has a **size**, and you can't put a value of a bigger type into a smaller variable, no matter what the actual size of the data is.
- When you're using **literal** values, use the F suffix to indicate a float (15.6F) and M for a decimal (36.12M).
- Use the **decimal type for money and currency**. Floating-point precision is…well, it's a little weird.
- There are a few types that C# knows how to **convert** automatically (an implicit conversion), like short to int, int to double, or float to double.
- When the compiler won't let you set a variable equal to a value of a different type, that's when you need to cast it. To **cast** a value (an explicit conversion) to another type, put the target type in parentheses in front of the value.
- Some keywords are **reserved** by the language and you can't name your variables with them. They're words (like `for`, `while`, `using`, `new`, and others) that do specific things in the language.
- A **parameter** is what you define in your method. An **argument** is what you pass to it.
- When you build your code in the IDE, it uses the **C# compiler** to turn it into an executable program.
- You can use methods on the static **Convert class** to convert values between different types.

Owen is constantly improving his game...

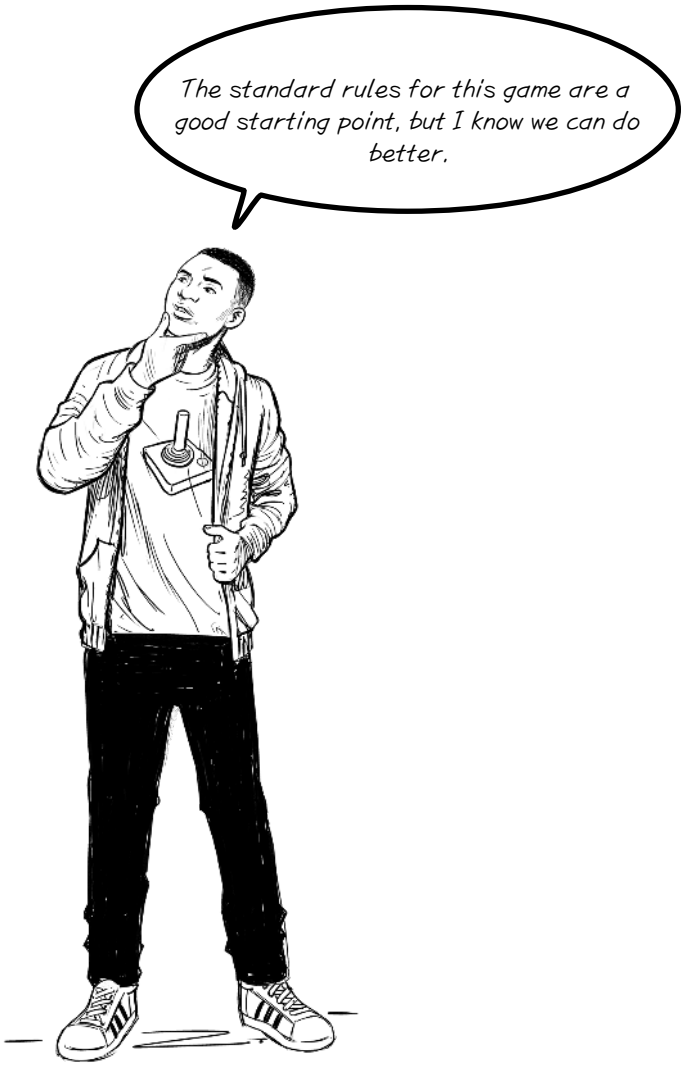
Good game masters are dedicated to creating the best experience they can for their players. Owen's players are about to embark on a new campaign with a brand-new set of characters, and he thinks a few tweaks to the formula that they use for their ability scores could make things more interesting.

When players fill out their character sheets at the start of the game, they follow these steps to calculate each of the ability scores for their character.

ABILITY SCORE FORMULA

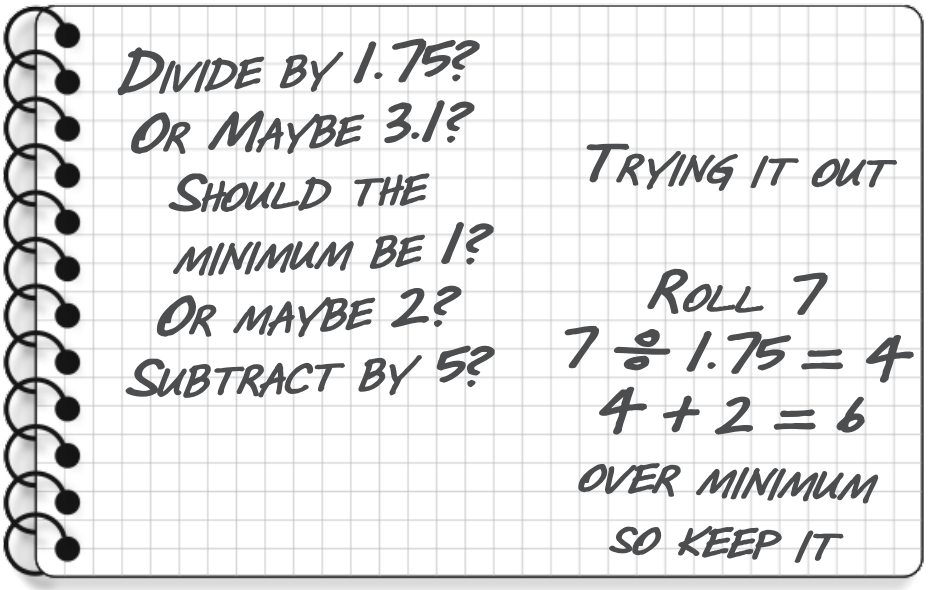
* Start with a 4d6 roll to get a number between 4 and 24
* Divide the roll result by 1.75
* Add 2 to the result of that division
* Round down to the nearest whole number
* If the result is too small, use the minimum value of 3

A "4d6 ROLL" means rolling four normal six-sided dice and adding up the results.

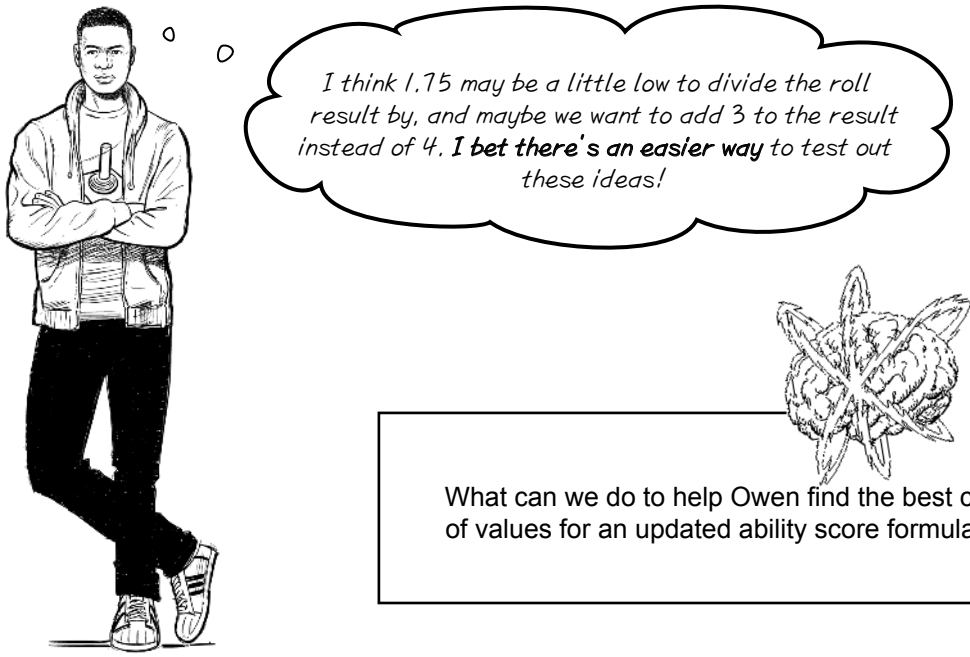


...but the trial and error can be time-consuming

Owen's been experimenting with ways to tweak the ability score calculation. He's pretty sure that he has the formula mostly right—but he'd really like to tweak the numbers.



Owen likes the overall formula: 4d6 roll, divide, subtract, round down, use a minimum value…but he's not sure that the actual numbers are right.



Brain Power

What can we do to help Owen find the best combination of values for an updated ability score formula?

Let's help Owen experiment with ability scores

In this next project, you'll build a console app that Owen can use to test his ability score formula with different values to see how they affect the resulting score. The formula has **four inputs**: the *starting 4d6 roll*, the *divide by* value that the roll result is divided by, the *add amount* value to add to the result of that division, and the *minimum* to use if the result is too small.

Owen will enter each of the four inputs into the app, and it will calculate the ability score using those inputs. He'll probably want to test a bunch of different values, so we'll make the app easier to use by asking for new values over and over again until he quits the app, keeping track of the values he used in each iteration and using those previous inputs as **default values** for the next iteration.

Here's the page from Owen's game master notebook with the ability score formula.

ABILITY SCORE FORMULA

- Start with a 4d6 roll to get a number between 4 and 24
- Divide the roll result by 1.75
- Add 2 to the result of that division
- Round down to the nearest whole number
- If the result is too small, use the minimum value of 3

This is what it looks like when Owen runs the app:

```
C:\Users\Public\source\repos'
Starting 4d6 roll [14]:
   using default value 14
Divide by [1.75]:
   using default value 1.75
Add amount [2]:
   using default value 2
Minimum [3]:
   using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [14]:
   using default value 14
Divide by [1.75]: 2.15
   using value 2.15
Add amount [2]: 5
   using value 5
Minimum [3]: 2
   using value 2
Calculated ability score: 11
Press Q to quit, any other key to continue
Starting 4d6 roll [14]: 21
   using value 21
Divide by [2.15]:
   using default value 2.15
Add amount [5]:
   using default value 5
Minimum [2]:
   using default value 2
Calculated ability score: 14
Press Q to quit, any other key to continue
```

The app prompts for the various values used to calculate the ability score. It puts a default value like [14] or [1.75] in square brackets. Owen can enter a value, or just hit Enter to accept a default value.

Here Owen is trying out new values: divide the roll result by 2.15 (instead of 1.75), add 5 (instead of 2) to the result of that division, and a minimum value of 2 (instead of 3). With an initial roll of 14, that gives an ability score of 11.

Now Owen wants to check those same values with a different starting 4d6 roll, so he enters 21 as the starting roll, presses Enter to accept the default values that the app remembered from the previous iteration, and gets an ability score of 14.

This project is a little larger than the previous console apps that you've built, so we'll *tackle it in a few steps*. First you'll Sharpen your Pencil to understand the code to calculate the ability score. Then you'll do an Exercise to write the rest of the code for the app. And finally, you'll Sleuth out a bug in the code. Let's get started!

Sharpen your pencil

We've built a class to help Owen calculate ability scores. To use it, you'll set its RollResult, DivideBy, AddAmount, and Minimum fields—or just leave the values set in their declarations—and call its CalculateAbilityScore method.

Create a new Console App project called AbilityScore and **add a class called AbilityScoreCalculator**. Enter all of the code into the class file. Uh-oh! There's one line of code that has a problem. **Circle the line of code** that causes a compiler error. Then **write down what you think you'll need to do to fix it**.

```
namespace AbilityScore;

internal class AbilityScoreCalculator
{
    public int RollResult = 14;
    public double DivideBy = 1.75;
    public int AddAmount = 2;
    public int Minimum = 3;
    public int Score;

    public void CalculateAbilityScore()
    {
        // Divide the roll result by the DivideBy field
        double divided = RollResult / DivideBy;

        // Add AddAmount to the result and round down
        int added = AddAmount += divided;

        // If the result is too small, use Minimum
        if (added < Minimum)
        {
            Score = Minimum;
        } else
        {
            Score = added;
        }
    }
}
```

If you're using VSCode, you can replace the "internal" access modifier with "public"—it won't make a difference in how the app works.

These fields are initialized with the values from the ability score formula. The app will use them to present default values to the user.

Here's a hint! Visual Studio will show you which line of code won't compile, and underline the specific part that has problems. Hover over that part and look closely at the error message.

After you **circle the line of code that won't compile**, write down what you need to do to fix the compiler error.

.....

.....

Fix the compiler error by adding a cast

Sharpen your pencil Solution

If you entered the code correctly, you should see a C# compiler error on this line of code:

```
int added = AddAmount += divided;
```

Here's the line of code to circle in the "Sharpen your pencil" exercise.

(field) int AbilityScoreCalculator.AddAmount

No examples or documentation available.

CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

This C# compiler error message is giving you a big hint—it looks like we forgot to cast a value.

Any time the C# compiler gives you an error, read it carefully. It often has a hint that can help you track down the problem. This error tells us exactly what went wrong: ***it can't convert a double to an int without a cast***. The `divided` variable is declared as a double, but C# won't allow you to add it to an int field like AddAmount because it doesn't know how to convert it. **So here's the answer** to the "Sharpen your pencil" question:

After you **circle the line of code that has problems**, look at the error and write down what you need to do to fix it.

The compiler error says it can't convert a double to an int, and asks if our code is missing a cast.

To fix it, we need to use (int) to cast the double to an int so += will be able to add the values.

When the C# compiler asks "are you missing a cast?" it's giving you a huge hint that you need to **cast the double variable `divided`** before you can add it to the int field AddAmount.

But this isn't the whole answer! There's still something wrong with that line of code. Can you spot it?

Add a cast to get the AbilityScoreCalculator class to compile...

Now that you know what the problem is, you can **add a cast** to fix the problematic line of code in AbilityScoreCalculator. The line that caused the error because **`AddAmount += divided` *returns a double value***. When you try to store a double value in an int variable like `added`, you'll get a "Cannot implicitly convert type" error.

You can fix it by **casting `divided` to an int**, so adding it to AddAmount returns another int. Modify that line of code to change `divided` to `(int)divided`:

```
int added = AddAmount += (int)divided;
```

Cast this!

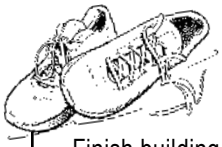
Adding that cast also *addresses an important part* of Owen's ability score formula:

* *Round down to the nearest whole number*

When you cast a double to an int, C# rounds it down—so for example `(int)19.7431D` gives us `19`. By adding that cast, you're making sure the score is rounded down, like Owen's formula asks for.

...but there's still a bug!

We're not quite done yet! You fixed the compiler error, so now the project builds. But even though the C# compiler will accept it, ***there's still a bug in the code***. So let's go ahead and fix it! In the next exercise, you'll use the AbilityScoreCalculator class as is, then you'll use it to sleuth out the bug.



Exercise

Finish building the console app that uses the AbilityScoreCalculator class. In this exercise, we'll give you the top-level statements for the console app. Your job is to write code for two methods: a method called ReadInt that reads user input and converts it to an int using int.TryParse, and a method called ReadDouble that does exactly the same thing except it parses doubles instead of int values.

Step 1: In this first step, you'll add top-level statements to your *Program.cs* file. Almost everything was used in previous projects. There's only one new thing—it calls the Console.ReadKey method:

```
char keyChar = Console.ReadKey(true).KeyChar;
```

Console.ReadKey reads a single key from the console. When you pass the argument **`true`**, it intercepts the input so that it doesn't get printed to the console. Adding **`.KeyChar`** causes it to return the key pressed as a **char**.

Delete the "Hello, World!" line from your *Program.cs* file and **add these top-level statements**:

```
using AbilityScore;

AbilityScoreCalculator calculator = new AbilityScoreCalculator();
while (true)
{
    calculator.RollResult = ReadInt(calculator.RollResult, "Starting 4d6 roll");
    calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
    calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
    calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
    calculator.CalculateAbilityScore();
    Console.WriteLine("Calculated ability score: " + calculator.Score);
    Console.WriteLine("Press Q to quit, any other key to continue");
    char keyChar = Console.ReadKey(true).KeyChar;
    if ((keyChar == 'Q') || (keyChar == 'q')) return;
}
```

Step 2: The code you wrote calls a method called ReadInt, so **add a static ReadInt method**. The ReadInt method takes two parameters: a string called **prompt** to display to the user, and an int called **defaultValue**. It writes the prompt to the console, followed by the default value in square brackets. Then it reads a line from the console and attempts to parse it with int.TryParse. If that returns true, return that value; otherwise, return the default value.

Here's the declaration:

```
static int ReadInt(int defaultValue, string prompt)
```

Calling `ReadInt("37", "What's the magic number?")` will cause the following prompt to be printed:

```
What's the magic number? [37]
```

There's a space at the end of that prompt. The user then types in a value and presses Enter. The method reads that line from the console and calls int.TryParse to try to parse it. If int.TryParse returns true, the method returns the result. If it returns false, the method returns defaultValue—in this case, 37.

Step 3: Generate and implement the ReadDouble method. ReadDouble is exactly like ReadInt, except that **it uses double.TryParse** instead of int.TryParse. The double.TryParse method works exactly like int.TryParse, except its **`out`** variable needs to be a double, not an int.

```
static double ReadDouble(double defaultValue, string prompt)
```

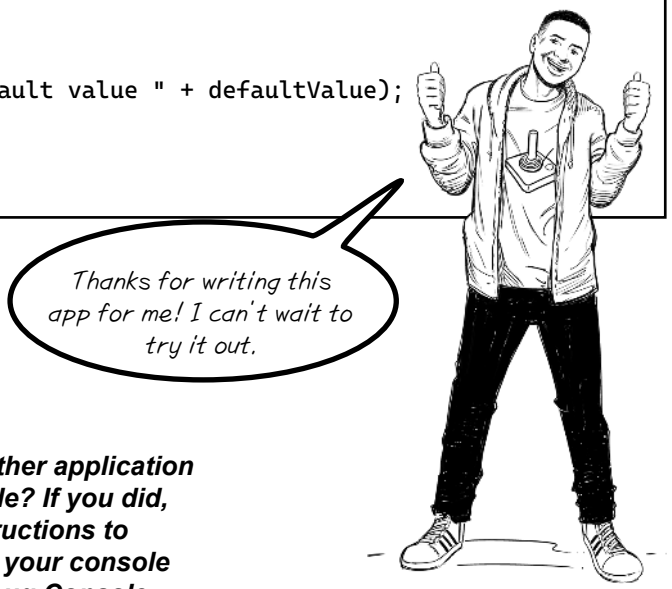
Exercise Solution

Here are the ReadInt and ReadDouble methods that display a prompt that includes the default value, read a line from the console, try to convert it to an int or a double, and either use the converted value or the default value, writing a message to the console with the value returned.

```
static int ReadInt(int defaultValue, string prompt)
{
    Console.Write(prompt + " [" + defaultValue + "]: ");
    string? line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    } else
    {
        Console.WriteLine("   using default value " + defaultValue);
        return defaultValue;
    }
}

static double ReadDouble(double defaultValue, string prompt)
{
    Console.Write(prompt + " [" + defaultValue + "]: ");
    string? line = Console.ReadLine();
    if (double.TryParse(line, out double value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    }
    else
    {
        Console.WriteLine("   using default value " + defaultValue);
        return defaultValue;
    }
}
```

Really take some time to understand how each iteration of the `while` loop in the top-level statements uses *fields to save the values* that the user entered, then uses them for the default values in the next iteration.

Try experimenting with the global namespace by removing "using AbilityScore;" from the top-level statements. What else do you need to do to change the code to build?



Try changing the csharp.debug.console setting to the external Terminal to run your app in an external terminal window. You might prefer debugging your apps that way!

Did you get a "cannot read keys when either application does not have a console" error in VSCode? If you did, go back to Chapter 1 and follow the instructions to change the C# debug console setting so your console app runs in the Terminal and not the Debug Console.

Here's the output from the app.

```
Starting 4d6 roll [14]: 18
   using value 18
Divide by [1.75]: 2.15
   using value 2.15
Add amount [2]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 13
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [2.15]: 3.5
   using value 3.5
Add amount [13]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [10]: 7
   using value 7
Minimum [3]:
   using default value 3
Calculated ability score: 12
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [12]: 4
   using value 4
Minimum [3]:
   using default value 3
Calculated ability score: 9
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [9]:
   using default value 9
Minimum [3]:
   using default value 3
Calculated ability score: 14
Press Q to quit, any other key to continue
```

That's strange. Owen entered 5 for the previous add amount, but the program is giving him 10 as a default option.

Again, the last amount Owen entered was 7, but it's giving 12 as a default option. Weird.

Where did this 9 number come from? Did we see it before? Can that give us a hint about what's causing this bug?

Something's wrong. It's supposed to remember the values I enter, but it doesn't always work.

Look! In the first iteration, I entered 5 for the add amount. It remembered all the other values just fine, but it gave me a default add amount of 10. That's the wrong result.

You're right, Owen. There's a bug in the code.

Owen wants to try out different values to use in his ability score formula, so we used a loop to make the app ask for those values over and over again.

To make it easier for Owen to just change one value at a time, we included a feature in the app that remembers the last values he entered and presents them as default options. We implemented that feature by keeping an instance of the AbilityScoreCalculator class in memory, and updating its fields in each iteration of the `while` loop.

But something's gone wrong with the app. It remembers most of the values just fine, but it remembers the wrong number for the "add amount" default value. In the first iteration Owen entered 5, but it gave him 10 as a default option. Then he entered 7, but it gave a default of 12. What's going on?

Brain Power

What steps can you take to track down the bug in the ability score calculator app?

The Case of the Operator Oddity

The debugger is like a detective's magnifying glass. It helps you spot even the smallest clues.

Let's do an investigation and see if we can apprehend the culprit, Sherlock Holmes style. ***Something*** is causing the bug, so let's use the debugger to identify suspects and retrace their steps.

The problem seems to be isolated to the "add amount" value, so let's start by looking for any line of code that touches the AddAmount field. Here's the line that uses the AddAmount field—put a breakpoint on it (and don't forget to use F5 to start your app with debugging if the breakpoint doesn't fire):

```
39  calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
40  calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
41  calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
```

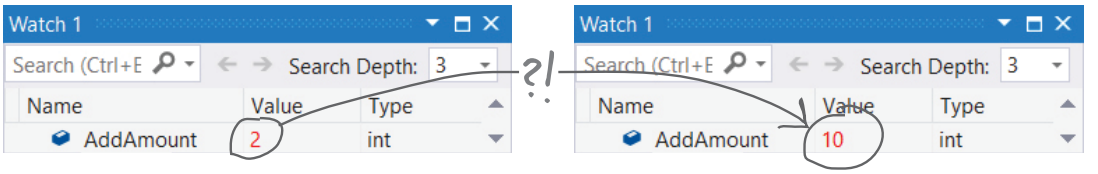
Here's another one in the AbilityScoreCalculator.CalculateAbilityScore method—breakpoint that suspect too:

```
20  // Add to the result
21  int added = AddAmount += (int)divided;
```

The trap is set. Let's see who springs it.

Now run your program. When your code hits the breakpoint, **select calculator.AddAmount and add a watch** (if you just right-click on AddAmount and choose Add Watch from the menu, it will only add a watch for AddAmount and not calculator.AddAmount). Does anything look weird there? We're not seeing anything unusual. It seems to read the value and update it just fine—that's probably not the issue. You can delete that breakpoint.

Continue running your program. When the breakpoint in AbilityScoreCalculator.CalculateAbilityScore is hit, **add a watch for AddAmount**. According to Owen's formula, this line of code is supposed to add AddAmount to the result of dividing the roll result. Now **step over** the statement and...



Wait, what?! AddAmount changed. But...but that's not supposed to happen—it's impossible! Right? As Sherlock Holmes said, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

It looks like we've sleuthed out the source of the problem. That statement is supposed to cast `divided` to an int to round it down to an integer, then add it to AddAmount and store the result in `added`. It also has an unexpected side effect: it's updating AddAmount with the sum because **the statement uses the += operator**, which returns the sum but assigns the sum to AddAmount.

Now we can finally fix Owen's bug—and get the REAL Sharpen answer

Now that you know what's happening, you can **fix the bug**—and it turns out to be a pretty small change. You just need to change the statement to use + instead of +=:

```
int added = AddAmount + (int)divided;
```

Change the += to a + to keep this line of code from updating the "added" variable and fix the bug.

And we can finally have the ***real*** answer to the "Sharpen your pencil" question in the first part of this project.

After you **circle the line of code that has problems**, look at the error and write down what you need to do to fix it.

First, it won't compile because AddAmount += divided is a double, so a cast needs to happen to assign it to an int. Second, it uses += and not +, which causes the line to update AddAmount.

there are no Dumb Questions

Q: I'm still not clear on the difference between the + operator and the += operator. How do they work, and why would I use one and not the other?

A: There are several operators that you can combine with an equals sign. They include += for adding, -= for subtracting, /= for dividing, *= for multiplying, and %= for remainder. Operators like + that combine two values are called **binary operators**. Some people find this name a little confusing, but "binary" refers to the fact that the operator combines two values—"binary" means "involving two things"—not that it somehow operates only on binary numbers.

With binary operators, you can do something called **compound assignment**, which means instead of this:

```
a = a + c;
```

you can do this:

```
a += c;
```

The += operator tells C# to add a + c and then store the result in a.

and it means the same thing. The compound assignment `x op= y` is equivalent to `x = x op y` (that's the technical way of explaining it). They do exactly the same thing.

Operators like += or *= that combine a binary operator with an equals sign are called *compound assignment operators*.

Q: But then how did the added variable get updated?

A: What caused confusion in the score calculator is that the **assignment operator = also returns a value**. You can do this:

```
int q = (a = b + c)
```

which will calculate `a = b + c` as usual. The = operator ***returns*** a value, so **it will update q with the result** as well. So:

```
int added = AddAmount += divided;
```

is just like doing this:

```
int added = (AddAmount = AddAmount +
divided);
```

which causes `AddAmount` to be increased by `divided`, but stores that result in `added` as well.

Q: Wait, what? The equals operator returns a value?

A: Yes, = returns the value being set. So in this code:

```
int first;
int second = (first = 4);
```

both `first` and `second` will end up equal to 4. Open up a console app and use the debugger to test this. It really works!

Sharpen your pencil

We learned about XML Documentation Comments (or XMLDoc) in Chapter 3. We added XMLDoc to the ReadInt method from your Ability Score Calculator app—but we left a lot of blanks for you to fill in. Go back to the section in Chapter 3 where we talked about XMLDoc and try filling in the missing parts of the XMLDoc.

```
/// <.......................>
/// ...........................................................................
/// </.......................>
/// <....................... name="defaultValue">
/// ...........................................................................
/// </.......................>
/// <....................... name="prompt">
/// ...........................................................................
/// </.......................>
/// <returns>
/// ...........................................................................
/// </returns>
static int ReadInt(int defaultValue, string prompt)
{
```

In the examples we showed you in Chapter 3, the opening <returns> tag was on the same line as the closing </returns> tag. In this exercise, we put them on separate lines to give you more space to describe what the method returns.

We aren't giving you a solution for this "Sharpen" pencil-and-paper exercise. Instead, use the responses that the AI chatbots give you in the next "Sens-AI" section—compare them against your answers to see how you did.

- We asked you to fill in six blanks for three pairs of opening and closing tags. Did you get them right?
- We gave you four blanks to describe what the method does, its parameters, and return values. Compare your responses with the ones the chatbots generated. Do you agree with their descriptions? Remember—AIs don't always get everything right. You wrote this code, so your descriptions might be more accurate than the AI's.

Sens-AI

Ask an AI chatbot to add comments to your code

Adding **comments** to your code is a great habit to get into. We've used comments throughout the first few chapters of this book to help you understand the code we've written. Comments are really valuable, for a few reasons:

- Every developer knows what it feels like to look at code they wrote a long time ago and have absolutely no idea what it does or how it works. This happens a lot when working on a larger app (like the ones you'll write later in the book). Comments help us remember what we were thinking when we wrote code.
- Sometimes you've got a particularly complex bit of code, like the event handler method for the buttons in the animal matching game from Chapter 1. Comments can make it easier for you to figure out what's going on in that code—which can be really valuable if you're trying to sleuth out a bug.
- Developers will often leave themselves comments like this:

  ```
  //TODO: Finish this piece of code
  ```

  as a reminder that there's still work left to do in one part of an app.

This is a great use of an AI chatbot as a learning tool. We've given you a lot of information about comments. Now you can use AI to learn more on your own.

These are three really good reasons that developers use comments, but there are many other ways—so many that there's no way we could possibly include them all in this book. Here's an opportunity to learn more on your own.

Open an AI chatbot and give it this prompt: ***Why do developers use comments?***

There are many answers to this question, because there are lots of reasons that developers use comments. Try giving this prompt to more than one AI chatbot, or asking it multiple times.

Create a prompt to add comments to your code

Start a new session with an AI chatbot and **give it the following prompt**:

```
Here are the contents of Program.cs:

<paste in the contents of your Program.cs file>

Here are the contents of AbilityScoreCalculator.cs:

<paste in the contents of the file with the AbilityScoreCalculator class>
```

Use AI chatbots to learn more about XML Documentation

In Chapter 3 we learned about XML Documentation (or XMLDoc), special comments with three slashes that you can use to document your methods, fields, and other class members. AI chatbots are really good at generating XMLDoc.

Start a new AI chatbot session, and give it exactly the same prompt you just gave it for the comments, except replace the first sentence of the prompt with this one, which asks it to generate XMLDoc where it's needed.

```
Add comments to the code for this C# console application, using XMLDoc
for the classes, methods, and fields.
```

Compare the response from the AI against your answers to the "Sharpen" exercise. Which descriptions are better?

Hey, kid! Wanna see something **WEIRD?**

Try this!

Try adding this `if/else` statement to a console app and build the solution:

```
if (0.1M + 0.2M == 0.3M) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

You'll see a green squiggle under the second `Console`—it's an **`Unreachable code detected`** warning. The C# compiler knows that 0.1 + 0.2 is always equal to 0.3, so the code will never reach the `else` part of the statement. Run the code—it prints `They're equal` to the console.

Next, **change the float literals to doubles** (remember, literals like 0.1 default to double):

```
if (0.1 + 0.2 == 0.3) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

That's really strange. The warning moved to the first line of the `if` statement. Try running the program. Hold on, that can't be right! It printed `They aren't equal` to the console. How is 0.1 + 0.2 not equal to 0.3?

Now do one more thing. Change 0.3 to 0.30000000000000004 (with 15 zeros between the 3 and 4). Now it prints `They're equal` again. So apparently 0.1D plus 0.2D equals 0.30000000000000004D.

Wait, what?!

So is that why I should only use the **decimal type for money**, and never use double for currency values?

Exactly. Decimal has a lot more precision than double or float, so it avoids the 0.30000000000000004 problem.

Some floating-point types—not just in C#, but in most programming languages!—can give you ***rare*** weird errors. This is so strange! How can 0.1 + 0.2 be 0.30000000000000004?

It turns out that some numbers *can't be exactly represented* as a double—it has to do with how they're stored as binary data (0s and 1s in memory). For example, .1D is not *exactly* .1. Try multiplying `.1D * .1D`—you get 0.010000000000000002, not 0.01. But `.1M * .1M` gives you the right answer. That's why floats and doubles are really useful for a lot of things (like positioning a GameObject in Unity). If you need more rigid precision—like for a financial app that deals with money—decimal is the way to go.

there are no Dumb Questions

Q: I'm still not clear on the difference between conversion and casting. Can you explain it a little more clearly?

A: "Conversion" is a general, all-purpose term for converting data from one type to another. Casting is a much more specific operation, with explicit rules about which types can be cast to other types, and what to do when the data for the value from one doesn't quite match the type it's being cast to. You just saw an example of one of those rules—when a floating-point number is cast to an int, it's rounded down by dropping any decimal value. You saw another rule earlier about wrapping for integer types, where a number that's too big to fit into the type it's being cast to is wrapped using the remainder operator.

Q: Hold on a minute. Earlier you had me "wrap" numbers myself using the mod function on my calculator app. Now you're talking about remainders. What's the difference?

A: Mod and remainder are very similar operations. For positive numbers they're exactly the same: A % B is the remainder when you divide B into A, so: 5 % 2 is the remainder of 5 ÷ 2, or 1. (If you're trying to remember how long division works, that just means that 5 ÷ 2 is equal to 2 × 2 + 1, so the rounded quotient is 2 and the remainder is 1.) But when you start dealing with negative numbers, there's a difference between mod (or modulus) and remainder. You can see for yourself: your calculator will tell you that –397 mod 17 = 11, but if you use the C# remainder operator you'll get –397 % 17 = –6.

If you're math oriented and want to challenge yourself, here's a good research topic: see if you can find an explanation for why mod acts differently on a calculator versus in C#. Try asking your favorite AI to help explain it.

Q: Owen's formula had me dividing two values and then rounding the result down to the nearest integer. How does that fit in with casting?

A: Let's say you have some floating-point values:

```
float f1 = 185.26F;
double d2 = .0000316D;
decimal m3 = 37.26M;
```

and you want to cast them to int values so you can assign them to int variables `i1`, `i2`, and `i3`. We know that those int variables can only hold integers, so your program needs to do *something* to the decimal part of the number.

So C# has a consistent rule: it drops the decimal and rounds down: `f1` becomes 185, `d2` becomes 0, and `m3` becomes 37. But don't take our word for it—write your own C# code that casts those three floating-point values to int to see what happens.

Q: Is there a way to make literals easier to read, like how we put commas in really long numbers?

A: Yes. You can use **underscores** to make literals in your code easier to read. If you want to store 8,675,309 in an int, you can add a _ for each comma:

```
int i = 8_675_309;
```

That works for other literal types, too, including byte, float, double, and decimal.

There's a whole web page dedicated to the 0.30000000000000004 problem! Check out https://0.30000000000000004.com to see examples in a lot of different languages.

The 0.1D + 0.2D != 0.3D example is an *edge case*, or a problem or situation that only happens under certain rare conditions, usually when a parameter is at one of its extremes (like a very big or very small number). If you want to learn more about it, there's a great article by Jon Skeet about how floating-point numbers are stored in memory in .NET: *https://csharpindepth.com/Articles/FloatingPoint*

↑ Jon gave us some amazing technical review feedback for the very first edition of this book, and that made a huge difference for us. Thanks so much, Jon!

Use reference variables to access your objects

When you create a new object, you use a `new` statement to instantiate it, like `new Guy()` in your program at the end of Chapter 3—the `new` statement created a new Guy object on the heap. You still needed a way to *access* that object, and that's where a variable like `joe` came in: `Guy joe = new Guy()`. Let's dig a little deeper into exactly what's going on there.

The `new` statement creates the instance, but just creating that instance isn't enough. ***You need a reference to the object***. So you created a **reference variable**: a variable of type Guy with a name, like `joe`. So `joe` is a reference to the new Guy object you created. Any time you want to use that particular Guy, you can reference it with the reference variable called `joe`.

When you have a variable that's an object type, it's a reference variable: a reference to a particular object. Let's just make sure we get the terminology right since we'll be using it a lot. We'll use the first two lines of the "Joe and Bob" program from the previous chapter:

Here's the heap before your code runs. Nothing there.

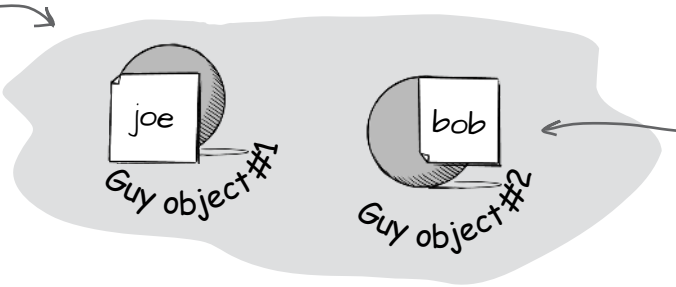
Creating a reference is like writing a name on a sticky note and sticking it to the object. You're using it to label an object so you can refer to it later.

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```

This is the reference variable.

This creates the object that it will refer to.

And here's the heap after this code runs. It has two objects, with the variable "joe" referring to one object and the variable "bob" referring to the other one.

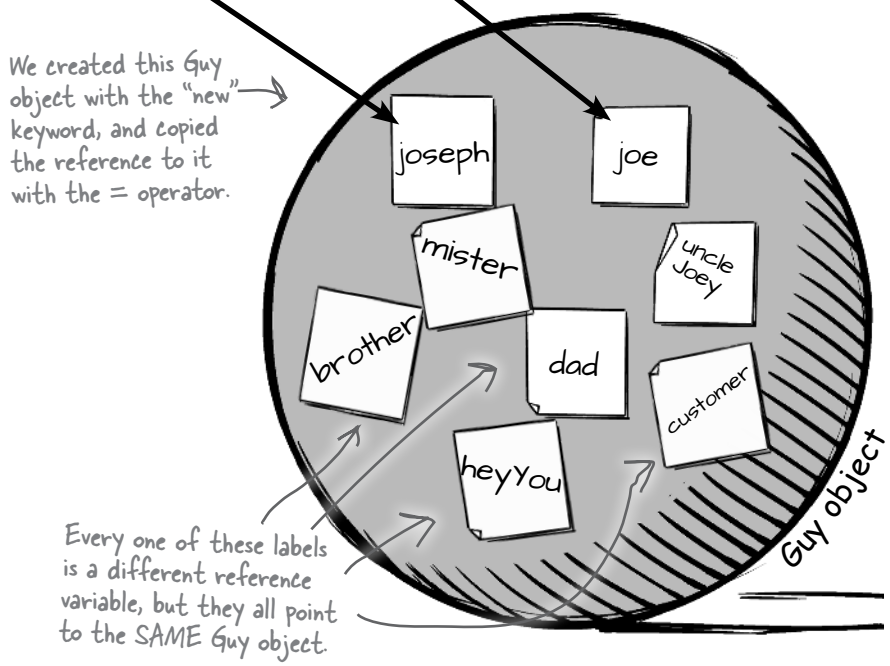


The ONLY way to reference this Guy object is through the reference variable called "bob."

References are like sticky notes for your objects

In your kitchen, you probably have containers of salt and sugar. If you switched their labels, it would make for a pretty disgusting meal—even though you changed the labels, the contents of the containers stayed the same. ***References are like labels.*** You can move labels around and point them at different things, but it's the **object** that dictates what methods and data are available, not the reference itself—and you can **copy references** just like you copy values.

```
Guy joe = new Guy();
Guy joseph = joe;
```



A reference is like a label that your code uses to talk about a specific object. You use it to access fields and call methods on an object that it points to.

We stuck a lot of sticky notes on that object! In this particular case, there are a lot of different references to this same Guy object—because a lot of different methods use it for different things. Each reference has a different name that makes sense in its context.

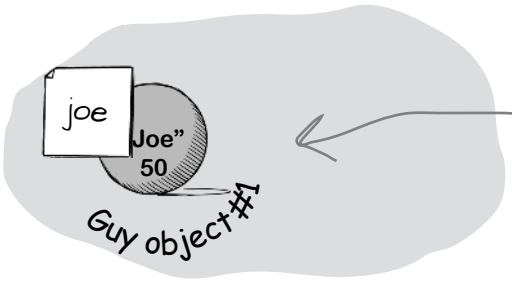
That's why it can be really useful to have ***multiple references pointing to the same instance***. So you could say `Guy dad = joe`, and then call `dad.GiveCash()` (that's what Joe's kid does every day). If you want to write code that works with an object, you need a reference to that object. If you don't have that reference, you have no way to access the object.

If there aren't any more references, your object gets garbage-collected

If all of the labels come off of an object, programs can no longer access that object. That means C# can mark the object for **garbage collection**. That's when C# gets rid of any unreferenced objects and reclaims the memory those objects took up for your program's use.

1. **Here's some code that creates an object.**
Just to recap what we've been talking about: when you use the `new` statement, you're telling C# to create an object. When you take a reference variable like `joe` and assign it to that object, it's like you're slapping a new sticky note on it.

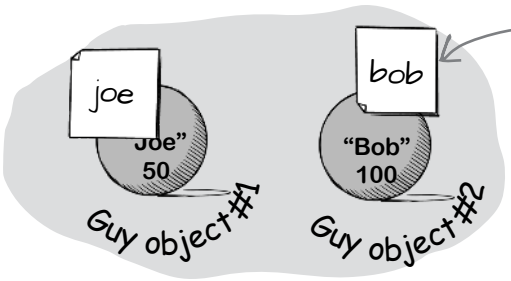
```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```



We used an object initializer to create this Guy object. Its Name field has the string "Joe," its Cash field has the int 50, and we put a reference to the object in a variable called "joe."

2. **Now let's create our second object.**
Once we do this we'll have two Guy object instances and two reference variables: one variable (`joe`) for the first Guy object, and another variable (`bob`) for the second.

```
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```



We created another Guy object and created a variable called "bob" that points to it. Variables are like sticky notes—they're just labels that you can "stick" to any object.

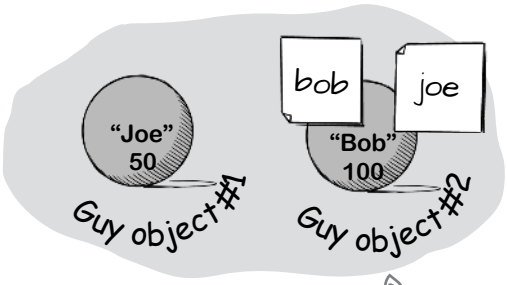
3 Let's take the reference to the first Guy object and change it to point to the second Guy object.

Take a really close look at what you're doing when you create a new Guy object. You're taking a variable and using the = assignment operator to set it—in this case, to a reference that's returned by the `new` statement. That assignment works because **you can copy a reference just like you copy a value**.

So let's go ahead and copy that value:

```
joe = bob;
```

That tells C# to make `joe` point to the same object that `bob` does. Now the `joe` and `bob` variables **both point to the same object**.

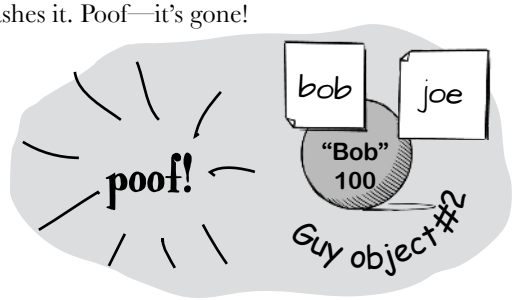


After the CLR (coming up in the "Garbage Collection Exposed" interview!) removes the last reference to the object, it marks it for garbage collection.

4 There's no longer a reference to the first Guy object...so it gets *garbage-collected*.

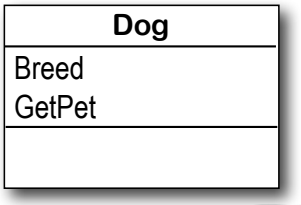
Now that `joe` is pointing to the same object as `bob`, there's no longer a reference to the Guy object it used to point to. So what happens? C# marks the object for garbage collection, and ***eventually*** trashes it. Poof—it's gone!

The CLR keeps track of all of the references to each object, and when the last reference disappears, it marks it for removal. But it might have other things to do right now, so the object could stick around for a few milliseconds—or even longer!



For an object to stay in the heap, it has to be referenced. Sometime after the last reference to the object disappears, so does the object.

```
public partial class Dog {
  public void GetPet() {
    Console.WriteLine("Woof!");
  }
}
```



Multiple references and their side effects

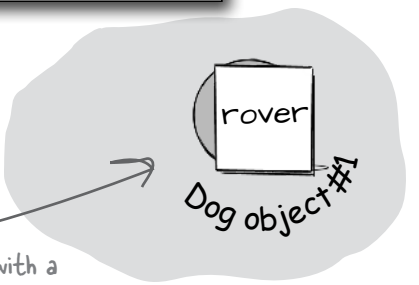
You've got to be careful when you start moving reference variables around. Lots of times, it might seem like you're simply pointing a variable to a different object. You could end up removing all references to another object in the process. That's not a bad thing, but it may not be what you intended. Take a look:

1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```

Objects: 1

References: 1



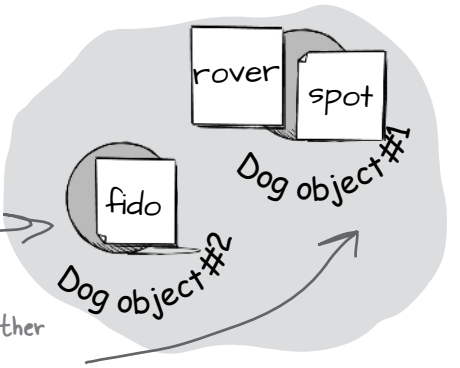
rover is a Dog object with a Breed field set to Greyhound.

2

```
Dog fido = new Dog();
fido.Breed = "Beagle";
Dog spot = rover;
```

Objects: 2

References: 3



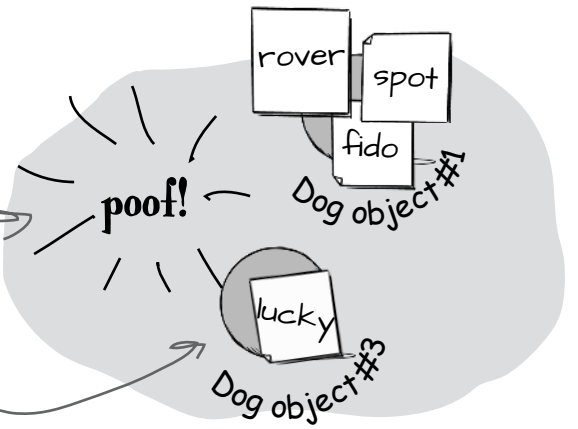
fido is another Dog object. spot is just another reference to the first object.

3

```
Dog lucky = new Dog();
lucky.Breed = "Dachshund";
fido = rover;
```

Objects: 2

References: 4



lucky is a third object. fido is now pointing to object #1. So, object #2 has no references. It's done as far as the program is concerned.

Sharpen your pencil

Now it's your turn. Here's one long block of code. Figure out how many objects and references there are at each stage. On the righthand side, draw a picture of the objects and sticky notes in the heap.

1
```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog bear = fido;
```

Objects:_____

References:_____

2
```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects:_____

References:_____

3
```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Objects:_____

References:_____

4
```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects:_____

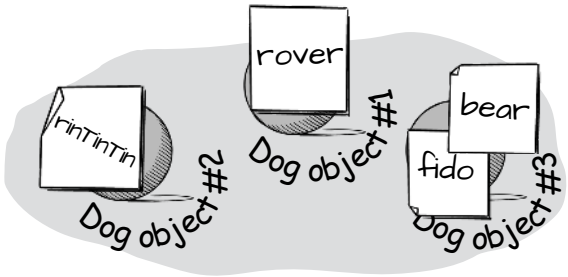
References:_____

5
```
charlie = laverne;
lucky = rinTinTin;
```

Objects:_____

References:_____

Sharpen your pencil Solution

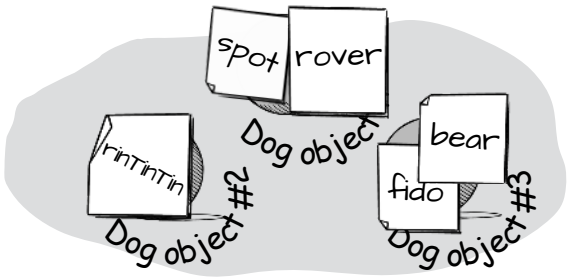


1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog bear = fido;
```

Objects: 3

References: 4



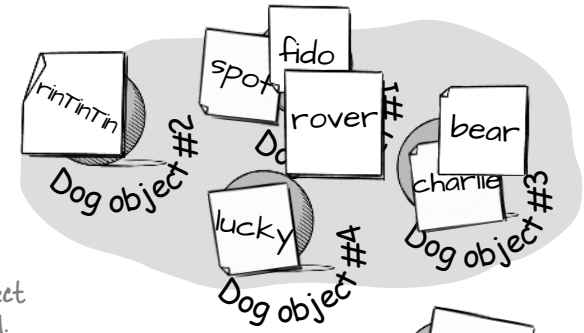
One new Dog object is created, but spot is the only reference to it. When spot is set to rover, that object goes away.

2

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects: 3

References: 5



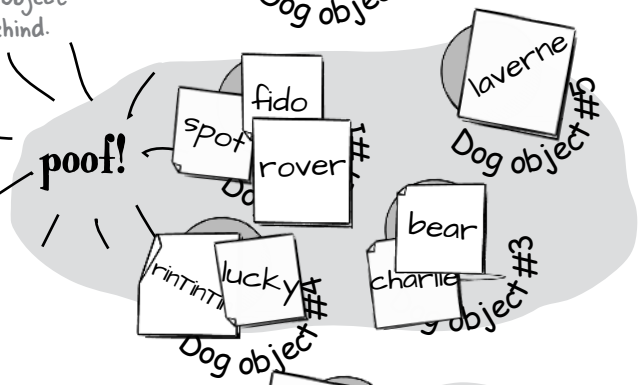
3

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

charlie was set to fido when fido was still on object #3. Then, after that, fido moved to object #1, leaving charlie behind.

Objects: 4

References: 7



Dog #2 lost its last reference, and it went away.

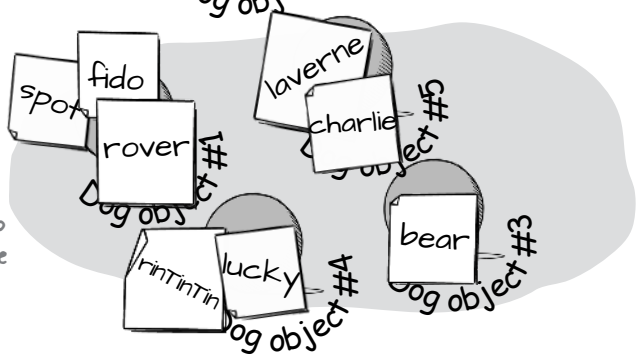
4

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects: 4

References: 8

When rinTinTin moved to lucky's object, the old rinTinTin object disappeared.



5

```
charlie = laverne;
lucky = rinTinTin;
```

Objects: 4

References: 8

Here the references move around, but no new objects are created. Setting lucky to rinTinTin did nothing because they already pointed to the same object.

Garbage Collection Exposed

This week's interview: The .NET Common Language Runtime

Head First: So, we understand that you do a pretty important job for us. Can you tell us a little more about what you do?

Common Language Runtime (CLR): In a lot of ways, it's pretty simple. I run your code. Any time you're using a .NET app, I'm making it work.

Head First: What do you mean by making it work?

CLR: I take care of the low-level "stuff" for you by doing a sort of "translation" between your program and the computer running it. When you talk about instantiating objects or doing garbage collection, I'm the one that's managing all of those things.

Head First: So how does that work, exactly?

CLR: Well, when you run a program on Windows, macOS, Linux, or most other operating systems, the OS loads machine language from a **binary**.

Head First: I'm going to stop you right there. Can you back up and tell us what machine language is?

CLR: Sure. A program written in machine language is made up of code that's executed directly by the CPU—and it's a whole lot less readable than C#.

Head First: If the CPU is executing the actual machine code, what does the OS do?

CLR: The OS makes sure each program gets its own process, respects the security rules, and provides APIs.

Head First: And for our readers who don't know what an API is?

CLR: An **API**—or **application programming interface**—is a set of methods provided by an OS, library, or program. OS APIs help you do things like work with the filesystem and interact with hardware. But they're often pretty difficult to use—especially for memory management—and they vary from OS to OS.

Head First: So back to your job. You mentioned a binary. What exactly is that?

CLR: A binary is a file that's (usually) created by a **compiler**, a program whose job it is to convert high-level language into low-level code like machine code. Windows binaries usually end with *.exe* or *.dll*.

Head First: But I'm guessing that there's a twist here. You said "low-level code like machine code"—does that mean there are other kinds of low-level code?

CLR: Exactly. I don't run the same machine language as the CPU. When you build your C# code, Visual Studio asks the C# compiler to create **Common Intermediate Language (CIL)**. That's what I run. C# code is turned into CIL, which I read and execute.

Head First: You talked about Windows binaries. But you also work on macOS. How does that work?

CLR: If you look in the folders created for your Visual Studio for Mac projects, you'll see lots of files that end with *.dll*. These are **managed .NET DLL files**, and they contain CIL code for the app. You can **run those apps from the command line** anywhere I'm installed! Try it out for yourself. Open a console window, go to the folder with the PickRandomCards project from Chapter 3, find the folder under bin/ that has files that end with *.dll*, and run this: `dotnet PickRandomCards.dll`.

Head First: You mentioned managing memory. Is that where garbage collection fits into all of this?

CLR: Yes! One useful thing that I do for you is manage your computer's memory by figuring out when your app is done with certain objects. When it is, I get rid of them for you to free up that memory. That's something programmers used to have to do themselves—but thanks to me, it's something that you don't have to be bothered with. You might not have known it at the time, but I've been making your job of learning C# a whole lot easier.

You can run your console apps from the command line. Find the DLL file underneath the bin/ folder and run it like this: `dotnet ProjectName.dll`—and this will work on any OS you can install .NET on—even Linux!

Exercise

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected. Here's what it will look like when your program runs.

You're going to build a new console app that has a class called Elephant.

Here's an example of the output of the program:

```
Press 1 for Lloyd, 2 for Lucinda, 3 to swap
You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.
```

The Elephant class has a WhoAmI method that writes these two lines to the console to display the values in the Name and EarSize fields.

Swapping the references causes the lloyd variable to call the Lucinda object's method, and vice versa.

Here's the class diagram for the Elephant class you'll need to create.

Elephant
Name = "" EarSize
WhoAmI

Swapping them again returns things to the way they were when the program started.

When you create your Elephant class, declare your Name field like this:

```
public string Name = "";
```

This sets the Name field to an empty string. Why do you think we're asking you to do that?

The CLR garbage-collects any object with no references to it. So here's a hint for this exercise: if you want to pour a cup of coffee into another cup that's currently full of tea, you'll need a third glass to pour the tea into...

Exercise

Your job is to create a console app with an Elephant class that matches the class diagram and uses its fields and methods to generate output that matches the example output.

1 **Create a new console app and add the Elephant class.**

Add an Elephant class to the project. Have a look at the Elephant class diagram—you'll need an int field called EarSize and a string field called Name. Add them, and make sure both are public. Then add a method called WhoAmI that writes two lines to the console to tell you the name and ear size of the elephant. Look at the example output to see exactly what it's supposed to print.

2 **Create two Elephant instances and a reference.**

Use object initializers to instantiate two Elephant objects:

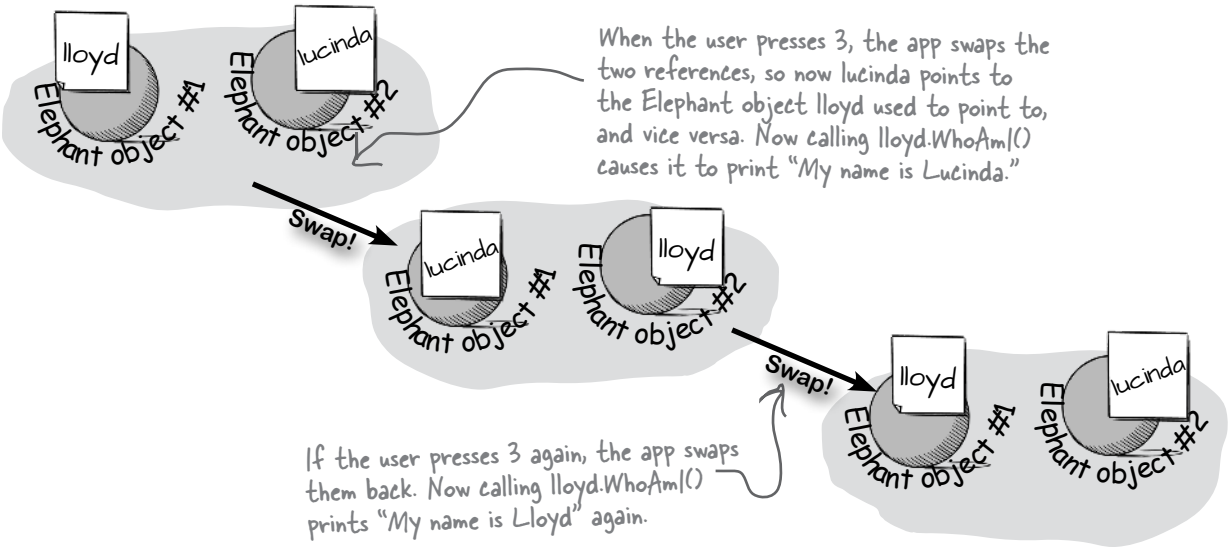
```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

3 **Call their WhoAmI methods.**

When the user presses 1, call lloyd.WhoAmI. When the user presses 2, call lucinda.WhoAmI. Make sure that the output matches the example. This is a good place to use `else if`, which you learned about in Chapter 3.

4 **Now for the fun part: swap the references.**

Here's the interesting part of this exercise. When the user presses 3, make the app execute code that ***exchanges the two references***. You'll need to write that method. After you swap references, pressing 1 should write Lucinda's message to the console, and pressing 2 should write Lloyd's message. If you swap the references again, everything should go back to normal.



Exercise Solution

There's more than one way to solve this exercise. If you came up with different code that works, that's great! Take the time to understand our solution.

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected.

Here's the Elephant class:

We asked you to initialize the Name field like this.

Elephant
Name = "" EarSize
WhoAmI

```
class Elephant
{
    public int EarSize;
    public string Name = "";
    public void WhoAmI()
    {
        Console.WriteLine("My name is " + Name + ".");
        Console.WriteLine("My ears are " + EarSize + " inches tall.");
    }
}
```

Here are the top-level statements for your *Program.cs* file:

```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };

Console.WriteLine("Press 1 for Lloyd, 2 for Lucinda, 3 to swap");
while (true)
{
    char input = Console.ReadKey(true).KeyChar;
    Console.WriteLine("You pressed " + input);
    if (input == '1')
    {
        Console.WriteLine("Calling lloyd.WhoAmI()");
        lloyd.WhoAmI();
    } else if (input == '2')
    {
        Console.WriteLine("Calling lucinda.WhoAmI()");
        lucinda.WhoAmI();
    } else if (input == '3')
    {
        Elephant holder;
        holder = lloyd;
        lloyd = lucinda;
        lucinda = holder;
        Console.WriteLine("References have been swapped");
    }
    else
    {
        return;
    }
    Console.WriteLine();
}
```

We used "else if" to check if the user entered 1, 2, or 3, and an else statement to exit if they entered anything else.

If you just point Lloyd to Lucinda, there won't be any more references pointing to Lloyd, and his object will be lost. That's why you need to have an extra variable (we called it "holder") to keep track of the Lloyd object reference until Lucinda can get there.

There's no "new" statement when we declare the "holder" variable because we don't want to create another instance of Elephant.

Two references mean TWO variables that can change the same object's data

Besides losing all the references to an object, when you have multiple references to an object, you can unintentionally change the object. In other words, one reference to an object may ***change*** that object, while another reference to that object has ***no idea*** that something has changed. Let's see how that works.

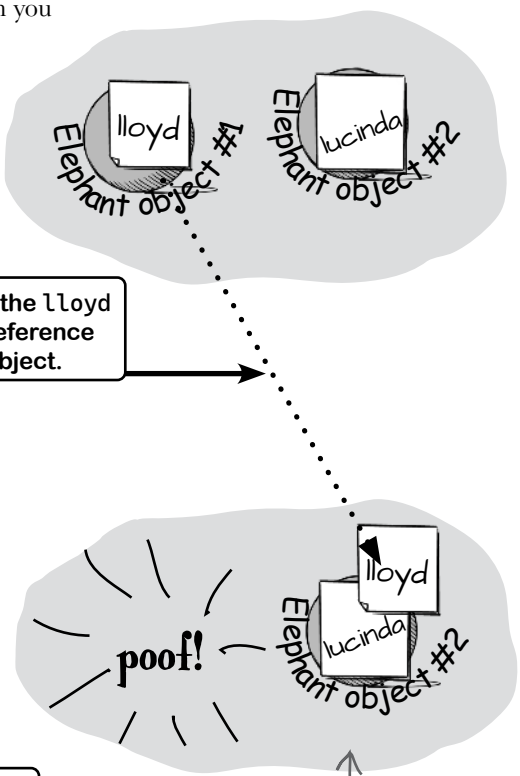
Do this!

Add one more `else if` block to your top-level statements. Can you guess what will happen once it runs?

```
else if (input == '3')
{
    Elephant holder;
    holder = lloyd;
    lloyd = lucinda;
    lucinda = holder;
    Console.WriteLine("References have been swapped");
}
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else
{
    return;
}
```

After this statement, both the `lloyd` and `lucinda` variables reference the SAME Elephant object.

This statement says to set EarSize to 4321 on whatever object the reference stored in the `lloyd` variable happens to point to.



Swapping these two sticky notes won't change anything because they're stuck to the same object.

And since the lloyd reference is no longer pointing to the first Elephant object, it gets garbage-collected...and there's no way to bring it back!

Now go ahead and run your program. Here's what you'll see:

```
You pressed 4
My name is Lucinda
My ears are 4321 inches tall.

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.
```

The program acts normally... until you press 4. Once you do that, pressing either 1 or 2 prints the same output—and pressing 3 to swap the references doesn't do anything anymore.

After you press 4 and run the new code that you added, both the **`lloyd`** and **`lucinda`** variables **contain the same reference** to the second Elephant object. Pressing 1 to call lloyd.WhoAmI prints exactly the same message as pressing 2 to call lucinda.WhoAmI. Swapping them makes no difference because you're swapping two identical references.

Objects use references to talk to each other

Elephant
Name EarSize
WhoAmI HearMessage SpeakTo

So far, you've seen forms talk to objects by using reference variables to call their methods and check their fields. Objects can call one another's methods using references too. In fact, there's nothing that a form can do that your objects can't do, because your form is just another object. When objects talk to each other, one useful keyword that they have is **this**. Any time an object uses the **this** keyword, it's referring to itself—it's a reference that points to the object that calls it. Let's see what that looks like by modifying the Elephant class so instances can call each other's methods.

1 **Add a method that lets an Elephant hear a message.**

Let's add a method to the Elephant class. Its first parameter is a message from another Elephant object. Its second parameter is the Elephant object that sent the message:

Do this!

```
public void HearMessage(string message, Elephant whoSaidIt) {
    Console.WriteLine(Name + " heard a message");
    Console.WriteLine(whoSaidIt.Name + " said this: " + message);
}
```

Here's what it looks like when it's called:

```
lloyd.HearMessage("Hi", lucinda);
```

We called **lloyd's** HearMessage method, and passed it two parameters: the string **"Hi"** and a reference to Lucinda's object. The method uses its **whoSaidIt** parameter to access the Name field of whatever elephant was passed in.

2 **Add a method that lets an Elephant send a message.**

Now let's add a SpeakTo method to the Elephant class. It uses a special keyword: **this**. That's a reference that lets an object get a reference to itself.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.HearMessage(message, this);
}
```

An Elephant's SpeakTo method uses the "this" keyword to send a reference to itself to another Elephant.

Let's take a closer look at what's going on.

When we call the Lucinda object's SpeakTo method:

```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

It calls the Lloyd object's HearMessage method like this:

```
whoToTalkTo.HearMessage("Hi, Lloyd!", this);
```

Lucinda uses whoToTalkTo (which has a reference to Lloyd) to call HearMessage.

this is replaced with a reference to Lucinda's object.

```
[a reference to Lloyd].HearMessage("Hi, Lloyd!", [a reference to Lucinda]);
```

3 Call the new methods.

Add one more **else if** block to the top-level statements to make the Lucinda object send a message to the Lloyd object:

```
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else if (input == '5')
{
    lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
}
else
{
    return;
}
```

Now run your program and press 5. You should see this output:

```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

The "this" keyword lets an object get a reference to itself.

Remember, if your app doesn't pause on the breakpoint, make sure you're starting the app with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu.

4 Use the debugger to understand what's going on.

Place a breakpoint on the statement that you just added:

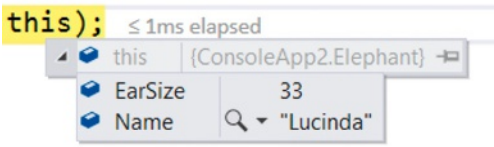
```
45  lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

Run your program and press 5.

When it hits the breakpoint, use Debug >> Step Into (F11) to step into the SpeakTo method.

Add a watch for Name to show you which Elephant object you're inside. You're currently inside the Lucinda object—which makes sense because the app just called lucinda.SpeakTo.

Hover over the **this** keyword at the end of the line and expand it. It's a reference to the Lucinda object.



Hover over **whoToTalkTo** and expand it—it's a reference to the Lloyd object.

The SpeakTo method has one statement—it calls whoToTalkTo.HearMessage. Step into it.

You should now be inside the HearMessage method. Check your watch again—now the value of the Name field is "Lloyd"—the Lucinda object called the Lloyd object's HearMessage method.

Hover over **whoSaidIt** and expand it. It's a reference to the Lucinda object.

Finish stepping through the code. Take a few minutes to really understand what's going on.

Arrays hold multiple values

Strings and arrays are different from the other data types you've seen in this chapter because they're the only ones without a set size (think about that for a bit).

If you have to keep track of a lot of data of the same type, like a list of prices or a group of dogs, you can do it in an **array**. What makes an array special is that it's a **group of variables** that's treated as one object. An array gives you a way of storing and changing more than one piece of data without having to keep track of each variable individually. When you create an array, you declare it just like any other variable, with a name and a type—except **the type is followed by square brackets**:

We saw arrays of strings in Chapter 3. Now let's take a deeper dive into how arrays work.

```
bool[] myArray;
```

Use the `new` keyword to create an array. Let's create an array with 15 bool elements:

```
myArray = new bool[15];
```

Use square brackets to set one of the values in the array. This statement sets the value of the fifth element of `myArray` to `false` by using square brackets and specifying the **index** 4. It's the fifth one because the first is `myArray[0]`, the second is `myArray[1]`, etc.:

```
myArray[4] = false;
```

You use the `new` keyword to create an array because it's an object—so an array variable is a kind of reference variable. In C#, arrays are *zero-based*, which means the first element has index 0.

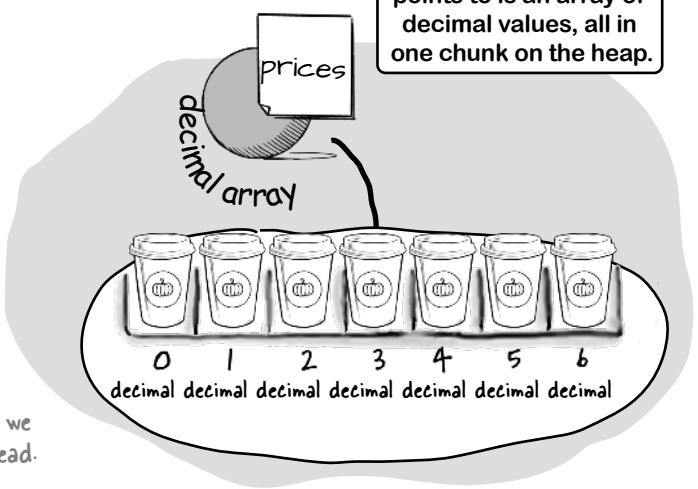
Use each element in an array like it's a normal variable

When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the `new` statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares and fills up an array—and what's happening in the heap when you do it. The first element in the array has an **index** of 0.

```
// declare a new 7-element decimal array
decimal[] prices = new decimal[7];
prices[0] = 12.37M;
prices[1] = 6_193.70M;

// we didn't set the element
// at index 2, it remains
// the default value of 0

prices[3] = 1193.60M;
prices[4] = 58_000_000_000M;
prices[5] = 72.19M;
prices[6] = 74.8M;
```

C# ignores underscores in your number literals, so we used commas to make our long numbers easier to read.

The `prices` variable is a reference, just like any other object reference. The object it points to is an array of decimal values, all in one chunk on the heap.



Arrays can contain reference variables

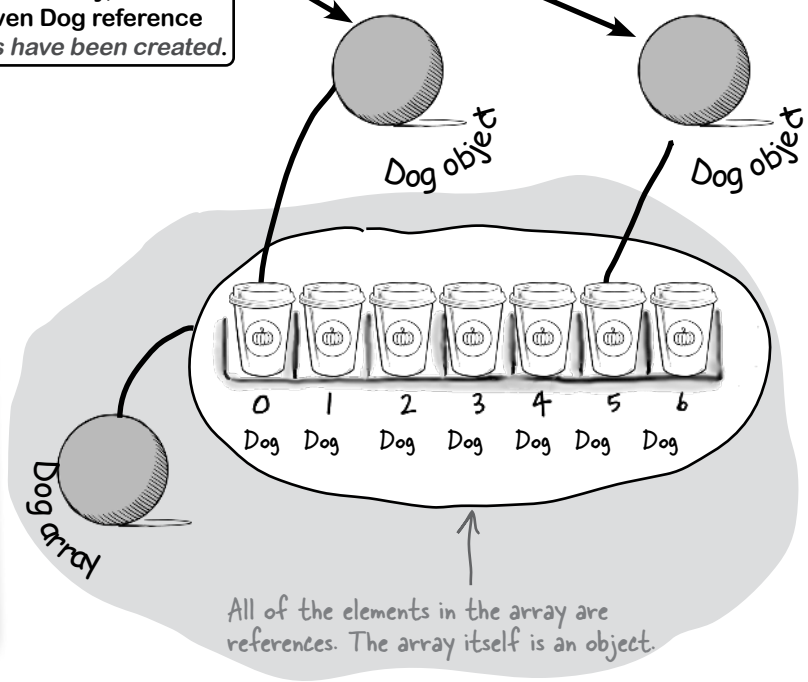
You can create an **array of object references** just like you create an array of numbers or strings. Arrays don't care what type of variable they store; it's up to you. So you can have an array of ints, or an array of Duck objects, with no problem.

Here's code that creates an array of seven `Dog` variables. The line that initializes the array only creates reference variables. Since there are only two `new Dog()` lines, only two actual instances of the Dog class are created.

```
// Declare a variable that holds an
// array of references to Dog objects
Dog[] dogs = new Dog[7];

// Create two new instances of Dog
// and put them at indexes 0 and 5
dogs[5] = new Dog();
dogs[0] = new Dog();
```

When you set or retrieve an element from an array, the number inside the brackets is called the index. The first element in the array has an index of 0.

The first line of code only created the array, not the instances. The array is a list of seven Dog reference variables—but *only two Dog objects have been created*.



An array's length

You can find out how many elements are in an array using its Length property. So if you've got an array called "prices," then you can use prices.Length to find out how long it is. If there are seven elements in the array, that gives you 7—which means the array elements are numbered 0 to 6.

there are no Dumb Questions

Q: I'm still not sure I get how references work.

A: References are the way you use all of the methods and fields in an object. If you create a reference to a Dog object, you can then use that reference to access any methods you've created for the Dog object. If the Dog class has (nonstatic) methods called Bark and Fetch, you can create a reference called spot, and then you can use that to call spot.Bark() or spot.Fetch(). You can also change information in the fields for the object using the reference (so you could change a Breed field using spot.Breed).

Q: Then doesn't that mean that every time I change a value through a reference I'm changing it for all of the other references to that object too?

A: Yes. If the `rover` variable contains a reference to the same object as `spot`, changing rover.Breed to "beagle" would make it so that spot.Breed was "beagle."

Q: Remind me again—what does `this` do?

A: `this` is a special variable that you can only use inside an object. When you're inside a class, you use `this` to refer to any field or method of that particular instance. It's especially useful when you're working with a class whose methods call other classes. One object can use it to send a reference to itself to another object. So if spot calls one of `rover`'s methods passing `this` as a parameter, it's giving `rover` a reference to the spot object.

Any time you've got code in an object that's going to be instantiated, the instance can use the special "this" variable that has a reference to itself.

Q: You keep talking about garbage-collecting, but what's actually doing the collecting?

A: Every .NET app runs inside the **Common Language Runtime** (or the *Mono Runtime* if you're running your apps on macOS, Linux, Android, or iOS, basically anywhere other than Windows, or if you're using Mono on Windows—but it works the same way, so we'll just talk about the CLR in general here).

The CLR does a lot of stuff, but there are two *really important things* the CLR does that we're concerned about right now. First, it **executes your code**—specifically, the output produced by the C# compiler. Second, it manages the memory that your program uses. That means it keeps track of all of your objects, figures out when the last reference to an object disappears, and frees up the memory that it was using. The folks on the .NET team at Microsoft have done an enormous amount of work making sure that it's fast and efficient.

Q: I still don't get that stuff about different types holding different-sized values. Can you go over that one more time?

A: Sure. The thing about variables is they assign a size to your number no matter how big its value is. So if you name a variable and give it a long type even though the number is really small (like, say, 5), the CLR sets aside enough memory for it to get really big. When you think about it, that's really useful. After all, they're called "variables" because they change all the time.

The CLR assumes you know what you're doing and you're not going to give a variable a type bigger than it needs. So even though the number might not be big now, there's a chance that after some math happens, it'll change. The CLR gives it enough memory to handle the largest value that type can accommodate.

Sharpen your pencil

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the `for` loop?

We're creating an array of seven Elephant references.

```
Elephant[] elephants = new Elephant[7];
elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };
```

Arrays start with index 0, so the first Elephant in the array is elephants[0].

```
Elephant biggestEars = elephants[0];
for (int i = 1; i < elephants.Length; i++)
{
    Console.WriteLine("Iteration #" + i);

    if (elephants[i].EarSize > biggestEars.EarSize)
    {
        biggestEars = elephants[i];
    }

    Console.WriteLine(biggestEars.EarSize.ToString());
}
```

This sets the biggestEars reference to the object that elephants[i] points to.

Be careful—this loop starts with the second element of the array (at index 1) and iterates six times until "i" is equal to the length of the array.

Fill in these values.

Iteration #1 biggestEars.EarSize = _____

Iteration #2 biggestEars.EarSize = _____

Iteration #3 biggestEars.EarSize = _____

Iteration #4 biggestEars.EarSize = _____

Iteration #5 biggestEars.EarSize = _____

Iteration #6 biggestEars.EarSize = _____

Sharpen your pencil Solution

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the `for` loop?

The for loop starts with the second Elephant and compares it to whatever Elephant biggestEars points to. If its ears are bigger, it points biggestEars at that Elephant instead. Then it moves to the next one, then the next one...by the end of the loop, biggestEars points to the one with the biggest ears.

```
Elephant[] elephants = new Elephant[7];
elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

Elephant biggestEars = elephants[0];
for (int i = 1; i < elephants.Length; i++)
{
    Console.WriteLine("Iteration #" + i);

    if (elephants[i].EarSize > biggestEars.EarSize)
    {
        biggestEars = elephants[i];
    }

    Console.WriteLine(biggestEars.EarSize.ToString());
}
```

Did you remember that the loop starts with the second element of the array? Why do you think that is?

The biggestEars reference keeps track of which Elephant we've seen so far has the biggest ears. Use the debugger to check this! Put your breakpoint here and watch biggestEars.EarSize.

Iteration #1 biggestEars.EarSize = 40

Iteration #2 biggestEars.EarSize = 42

Iteration #3 biggestEars.EarSize = 42

Iteration #4 biggestEars.EarSize = 44

Iteration #5 biggestEars.EarSize = 44

Iteration #6 biggestEars.EarSize = 45

null means a reference points to nothing

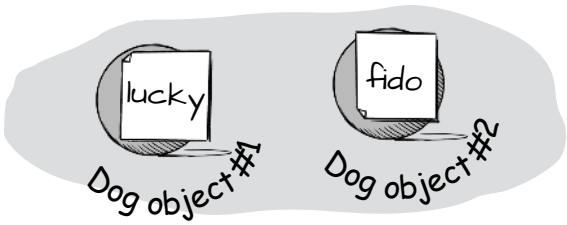
There's another important keyword that you'll use with objects. When you create a new reference and don't set it to anything, it has a value. It starts off set to `null`, which means **it's not pointing to any object at all**. Let's have a closer look at this:

The default value for any reference variable is `null`. Since we haven't assigned a value to `fido`, it's set to `null`.

```
Dog fido;
Dog lucky = new Dog();
```

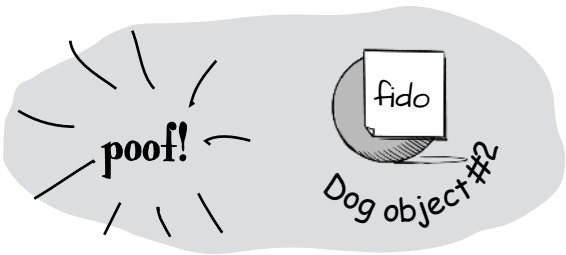
Now `fido` is set to a reference to another object, so it's not equal to `null` anymore.

```
fido = new Dog();
```



Once we set `lucky` to `null` it no longer points to its object, so it gets marked for garbage collection.

```
lucky = null;
```



*Would I ever **really** use "null" in a program?*

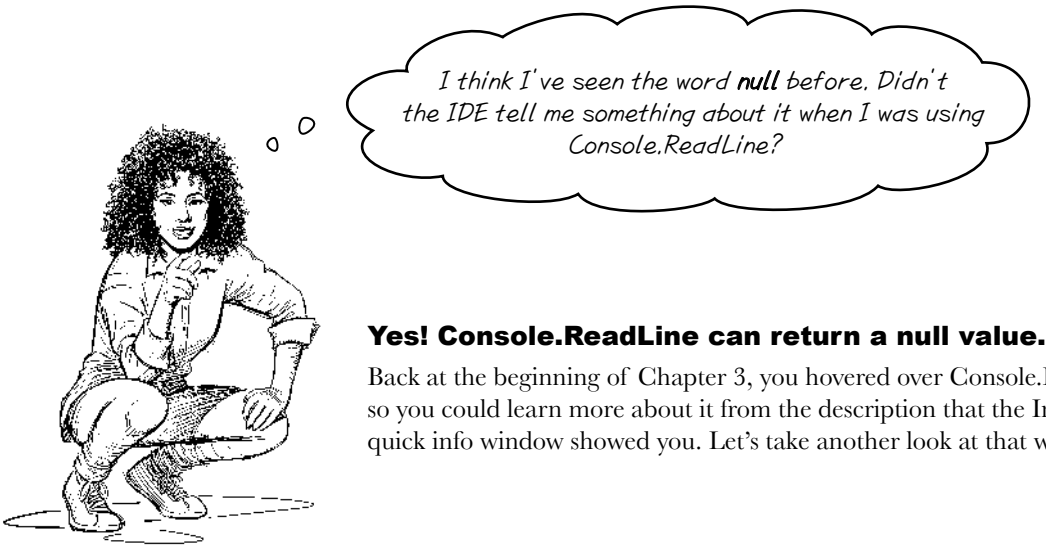
Yes. The null keyword can be useful.

There are a few ways you see `null` used in typical programs. The most common way is making sure a reference points to an object:

```
if (lloyd == null) {
```

That test will return `true` if the `lloyd` reference is set to `null`.

Another way you'll see the `null` keyword used is when you want your object to get garbage-collected. If you've got a reference to an object and you're finished with the object, setting the reference to `null` will immediately mark it for collection (unless there's another reference to it somewhere).



Yes! Console.ReadLine can return a null value.

Back at the beginning of Chapter 3, you hovered over Console.ReadLine so you could learn more about it from the description that the IntelliSense quick info window showed you. Let's take another look at that window:

```
string? line = Console.ReadLine();
```

string? Console.ReadLine()
Reads the next line of characters from the standard input stream.

Returns:
The next line of characters from the input stream, or null if no more lines are available.

Exceptions:
IOException
OutOfMemoryException
ArgumentOutOfRangeException

GitHub Examples and Documentation

Console.ReadLine will return the next line of characters that it reads. If there are no more lines, then it returns null.

Console.ReadLine returns a null when there are no lines available

You've been running your apps in Visual Studio and typing input using the keyboard. But you can also run them from the command line. In Windows, there's an executable in the bin\Debug folder. You can use this command to run your app from the project folder:

```
C:\Users\Public\source\repos\ConsoleApp1\ConsoleApp1>dotnet run
Hello, World!
```

Make sure you run from inside the project folder that has the .csproj file, not the solution folder that contains it.

You can also use your operating system's pipe commands like << or < or | to send input to your app from a file or the output of another console app. When you do this, Console.ReadLine needs a way to tell your app that it hit the end of the file—and that's when it returns null.

But there's still one issue: ***what does your app do when Console.ReadLine returns null?***

Use the string? type when a string might be null

You've been using two different (but related!) types to hold text values. First, there's the string **type**, like you used for the Name field in the Elephant class:

```
public string Name = "";
```

Then there's the **string? type**, like the type returned by Console.ReadLine or which int.TryParse takes as its first parameter, like you used in Owen's ability score calculator app:

```
string? line = Console.ReadLine();
if (int.TryParse(line, out int value))
```

The difference is that in the Elephant class ***the Name field is never null***. That's why we asked you to initialize the Name field in your Elephant class.

What do you think would happen if you didn't initialize the Name field in the Elephant class?

Do this!

Change the field declaration in the Elephant class so it doesn't initialize it to an empty string:

```
public string Name;
```

(field) string Elephant.Name

CS8618: Non-nullable field 'Name' must contain a non-null value when exiting constructor. Consider declaring the field as nullable.

Show potential fixes (Alt+Enter or Ctrl+.)

Visual Studio gives you a warning that has to do with null values, and asks you to consider declaring the field as nullable. That's what the string? type is—a **nullable string**.

You can make the error disappear by changing the Name field to a nullable string? instead of a string:

```
public string? Name;
```

Now your app builds again, and runs exactly the same way as it did before.

Use the debugger to step through the app, and add a watch for the holder variable. Its value is null until it runs this statement: `holder = lloyd;`

int.TryParse takes a string? parameter

So what does your app do if Console.ReadLine returns null?

Luckily, int.TryParse also takes a `string?` value, so if your app gets to the end of the input and Console.ReadLine returns null, int.TryParse will just return false—so the app will work just fine, and when it gets a null value it will treat it the way it treats any other value that can't be parsed.

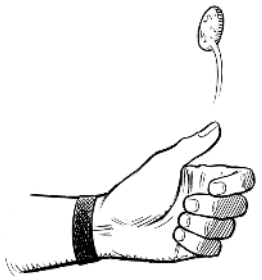
Visual Studio is smart enough to check for possible places where a value can be null. You can avoid that problem by making sure all of your reference variables are initialized.

Tabletop Games

There's a rich history to tabletop games—and, as it turns out, a long history of tabletop games influencing video games, at least as early as the very first commercial role-playing game.

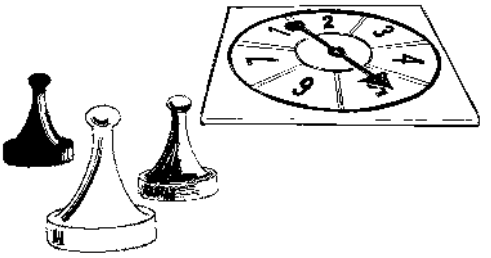
- The first edition of Dungeons and Dragons (D&D) was released in 1974, and that same year games with names like "dungeon" and "dnd" started popping up on university mainframe computers.
- You've used the Random class to create numbers. The idea of games based on random numbers has a long history—for example, tabletop games that use dice, cards, spinners, and other sources of randomness.
- We saw in Chapter 3 how a paper prototype can be a valuable first step in designing a video game. Paper prototypes have a strong resemblance to tabletop games. In fact, you can often turn the paper prototype of a video game into a playable tabletop game, and use it to test some game mechanics.
- You can use tabletop games—especially card games and board games—as learning tools to understand the more general concept of game mechanics. Dealing, shuffling, dice rolling, rules for moving pieces around the board, use of a sand timer, and rules for cooperative play are all examples of mechanics.
- The mechanics of Go Fish include dealing cards, asking another player for a card, saying "Go Fish" when asked for a card you don't have, determining the winner, etc. We're going to actually build a Go Fish game later in the book, so take a minute and read the rules: *https://en.wikipedia.org/wiki/Go_Fish#The_game*.

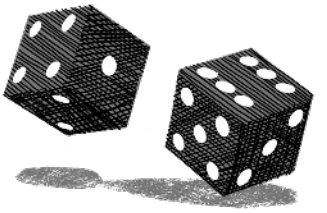
If you've never played Go Fish, take a few minutes and read the rules. We'll use them later in the book!



Even if we're not writing code for video games, there's a lot we can learn from tabletop games.

A lot of our programs depend on **random numbers**. For example, you've already used the Random class to create random numbers for several of your apps. Most of us don't actually have a lot of real-world experience with genuine random numbers… except when we play games. Rolling dice, shuffling cards, spinning spinners, flipping coins…these are all great examples of **random number generators**. The Random class is .NET's random number generator—you'll use it in many of your programs, and your experience using random numbers when playing tabletop games will make it a lot easier for you to understand what it does.





A Random Test Drive

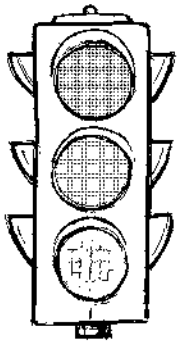
You'll be using the **Random class** throughout the book, so let's get to know it better by kicking its tires and taking it for a spin. Fire up Visual Studio and follow along—and make sure you run your code multiple times, since you'll get different random numbers each time.

1. **Create a new console app**—all of this code will go in the top-level statements. `Random.Shared` returns an **instance of the Random class**. Start by using it to generate a random int:

```
int randomInt = Random.Shared.Next();
Console.WriteLine(randomInt);
```

Specify a **maximum value** to get random numbers from 0 up to—but not including—the maximum value. A maximum of 10 generates random numbers from 0 to 9:

```
int zeroToNine = Random.Shared.Next(10);
Console.WriteLine(zeroToNine);
```



2. Now **simulate the roll of a die**. You can specify a minimum and maximum value. A minimum of 1 and maximum of 7 generates random numbers from 1 to 6:

```
int dieRoll = Random.Shared.Next(1, 7);
Console.WriteLine(dieRoll);
```

3. The **NextDouble method** generates random double values. Hover over the method name to see a tooltip—it generates a floating-point number from 0.0 up to 1.0:

```
double randomDouble = Random.Shared.NextDouble();
```

double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.

You can **multiply a random double** to generate much larger random double values. So if you want a random double value from 1 to 100, multiply the random double by 100:

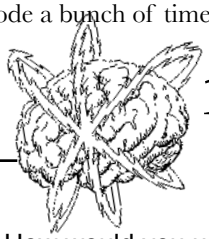
```
Console.WriteLine(randomDouble * 100);
```

Use **casting** to convert the random double to other types. Try running this code a bunch of times—you'll see tiny precision differences in the float and decimal values.

```
Console.WriteLine((float)randomDouble * 100F);
Console.WriteLine((decimal)randomDouble * 100M);
```

4. Use a maximum value of 2 to **simulate a coin toss**. That generates a random value of either 0 or 1. Use the **Convert class**, which has a static ToBoolean method that will convert it to a Boolean value:

```
int zeroOrOne = Random.Shared.Next(2);
bool coinFlip = Convert.ToBoolean(zeroOrOne);
Console.WriteLine(coinFlip);
```



Brain Power

How would you use Random to choose a random string from an array of strings?

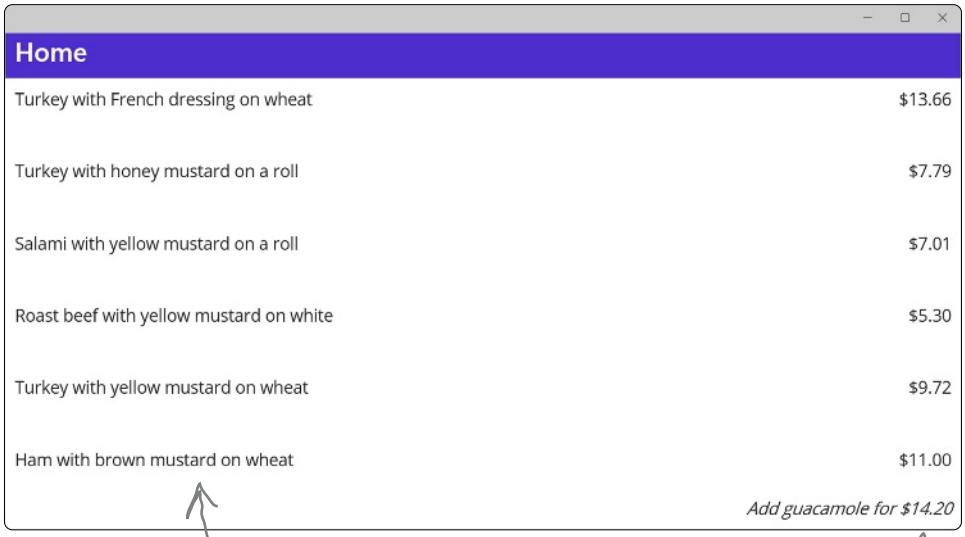
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. What he doesn't have is a menu! Can you build a program that makes a new random menu for him every day? You definitely can…with a **new MAUI app**, some arrays, your handy random number generator, and a couple of new, useful tools. Let's get started!

Here's the app you'll build. It creates a menu with six random sandwiches. Each sandwich has a protein, a condiment, and a bread, all chosen at random from a list. Every sandwich is given a random price, and there's a special random price at the bottom to add guacamole on the side.

Welcome to Sloppy Joe's, hon. The meat's nice and fresh! What can I getcha?

Sloppy Joe needs a new menu every day. Your app will generate random sandwiches and prices for him.



Each sandwich is generated by choosing a random protein, random condiment, and random bread from arrays.

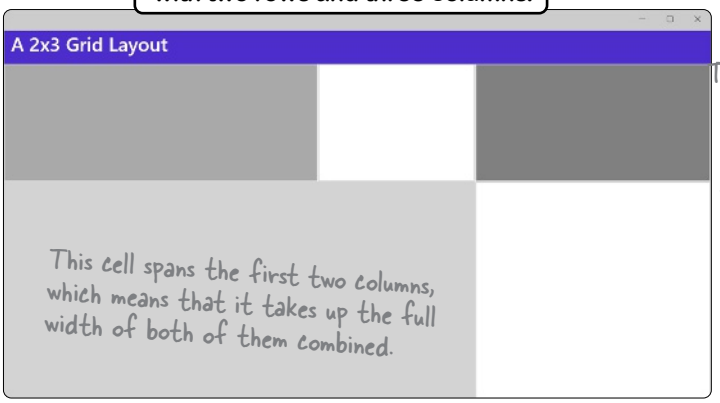
The prices are random numbers between 5.00 and 14.99.

Sloppy Joe's menu app uses a Grid layout

A **Grid control** contains other controls, and defines a set of rows and columns to lay out those controls.

You've used other layout controls: you've used VerticalStackLayout controls to stack Button, Label, and other controls in your apps on top of each other. You used a HorizontalStackLayout control in Chapter 2 for your bird picker. And in the Animal Matching Game project, your VerticalStackLayout contained a FlexLayout that arranged the buttons so they stacked horizontally, flowing into rows as the window size changed.

Here's an example of a Grid layout with two rows and three columns.



This is a row.

This cell spans the first two columns, which means that it takes up the full width of both of them combined.

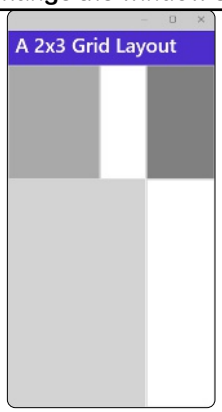
The second row height is twice as tall as the first row.

The first column is twice as wide as the second.

The second column.

The third column is 1.5 times as wide as the second.

The layout adjusts as you change the window size.



The Grid preserves the row and column proportions when you change the size and shape of the page, which is really useful when you want your app to run on devices with different screen sizes.

Watch it!

A Grid control is for *layouts*, not data.

When most of us see something that contains "rows" and "columns," we think of tables of data, like spreadsheets or HTML tables. That's not what a Grid control is all about.

*The Grid control is for **laying out content**. Its job is to contain other controls, and give you a way to design more interesting or intricate layouts than you get with stack panels, in a way that works well with different window sizes or on mobile devices.*

Grid controls

The Grid control contains other controls, and works just like the other layout controls to contain **child controls** (the other controls nested inside it). There's an opening `<Grid>` tag and a closing `</Grid>` tag, and the tags for all of the child controls are between them.

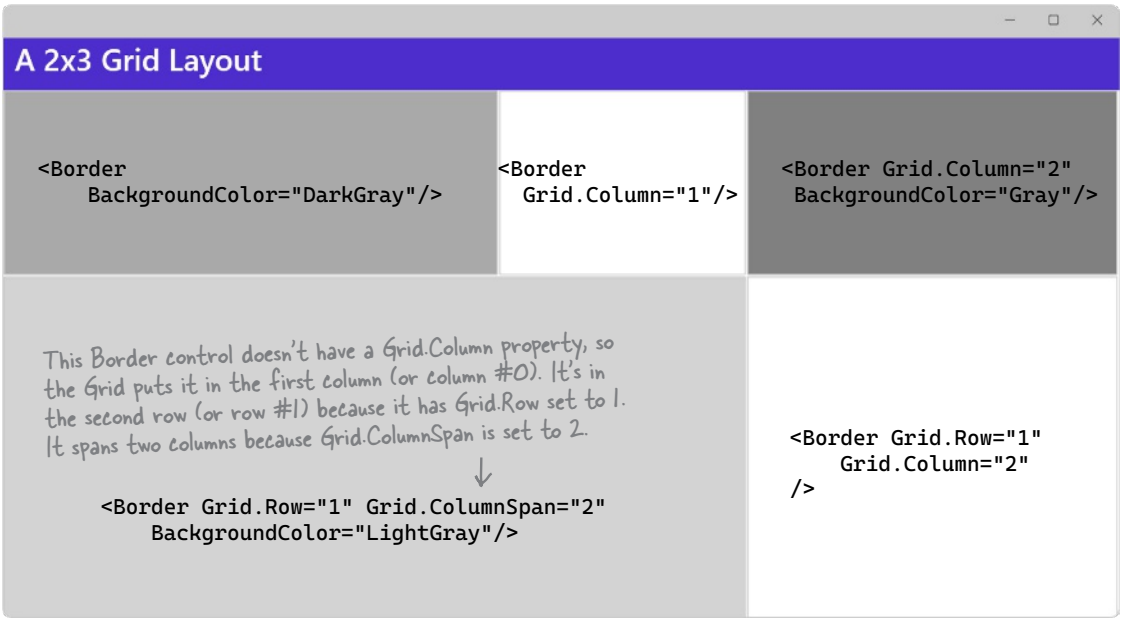
Cells in a grid are invisible—their only purpose is to determine where the child controls are displayed on the page. We used **Border controls** to make the grid visible. A Border control draws a border around a child control nested inside it:

```
<Border>
    <Label Text="I have a border!"/>
</Border>
```

A Border can only contain one child control. In the app below we didn't nest any controls inside the Borders—we just took advantage of the fact that each Border fills up the entire cell. We used the Border control's BackgroundColor property to make some of the cells in the grid darker.

Use Grid properties to put a control in a cell

The rows and columns in a Grid are numbered starting with 0. To put a child control in a specific row and column, use the Grid.Row and Grid.Column properties. For example, putting `<Border Grid.Row="1" Grid.Column="2" />` between Grid tags will make the Grid place the border in the second row and third column. You can also make a control span multiple rows or columns using the Grid.RowSpan and Grid.ColumnSpan properties.



Define the rows and columns for a Grid

The Grid control XAML has sections to define rows and columns. Each row or column can either have proportional sizes—for example, column 3 is twice as wide as column 2 and three times as wide as column 1—or absolute sizes in device-independent pixels.

The row and column definitions are in special sections inside the **`<Grid>`** tag. The row definitions are inside a <Grid.RowDefinitions> section, and the column definitions are inside a <Grid.ColumnDefinitions> section.

Here's the complete XAML for the app that we've been showing you. **Create a .NET MAUI app** called GridExample and add this XAML code (and delete the OnCounterClicked method in `MainPage.xaml.cs`).

Do this!

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridExample.MainPage">

    <ScrollView>

        <Grid>

            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="2*"/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2*"/>
                <ColumnDefinition/>
                <ColumnDefinition Width="1.5*"/>
            </Grid.ColumnDefinitions>

            <Border BackgroundColor="DarkGray"/>
            <Border Grid.Column="1" />
            <Border Grid.Column="2" BackgroundColor="Gray"/>

            <Border Grid.Row="1" Grid.ColumnSpan="2" BackgroundColor="LightGray"/>
            <Border Grid.Row="1" Grid.Column="2" />

        </Grid>

    </ScrollView>

</ContentPage>
```

The app has two rows, so the Grid.RowDefinitions section contains two `RowDefinition` tags. The second row height is twice as tall as the first row, so we added the `Height="2*"` property to the second `RowDefinition` tag to make it twice as tall.

The Grid.ColumnDefinitions section has three `ColumnDefinition` tags, one for each of the three columns. The first column is twice as wide as the second, so it has `Width="2*"`. The third column is 1.5 times as wide, so it has `Width="1.5*"`.

Here are the child Border controls we just showed you.

Row heights and column widths

When you use a value like 2* in a RowDefinition.Height or ColumnDefinition.Width property, you're choosing a proportional width, which means they're proportional to each other. You'll get the same results setting the first row to 6* and the second row to 12* because the proportions are still the same: the second row is still twice as big as the first row.

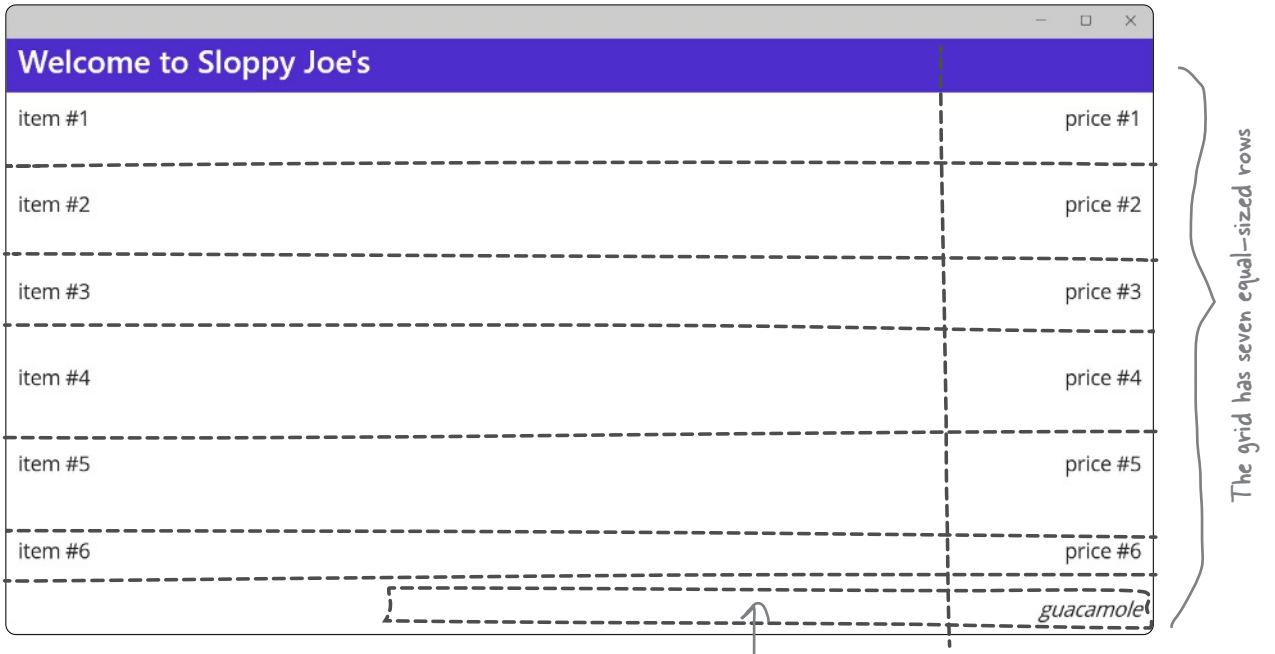
You can also set a row width or column height to an absolute value like 100, which will cause it to be sized in device-independent pixels. If all the rows and columns are proportional, the grid will fill up the page. If you set an absolute width or height, it could end up larger than the page, which is why it's a good idea to nest the Grid inside a ScrollView.

Create the Sloppy Joe's menu app and set up the grid

Create a new .NET MAUI app and name it SloppyJoe. The first thing you'll do is create the XAML for the app. Here's how it will work:

```
<ContentPage>
    <ScrollView>
        <Grid Margin="10">
```

The grid has two columns. Column 1 is 5 times wider than column 2.




```
        </Grid>
    </ScrollView>
</ContentPage>
```

Each of the cells in the grid contains a Label control...

...except for the Label with the guacamole price, which fills up the whole row by spanning two cells.

We'll give you all of the XAML for the app. But before we do, try editing the *MainPage.xaml* file and creating the XAML for the page on your own. Can you use the app we just gave you as an example to create the row and column definitions yourself?

See how far you can get, then compare it with our XAML.

Here's the XAML for the app

Take your time and go through it line by line to make sure you understand how its grid works.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SloppyJoe.MainPage">

    <ScrollView>

        <Grid Margin="10">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="5*"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>

            <Label x:Name="item1" FontSize="18" Text="item #1" />
            <Label x:Name="price1" FontSize="18" HorizontalOptions="End"
                   Grid.Column="1" Text="price #1"/>

            <Label x:Name="item2" FontSize="18" Grid.Row="1" Text="item #2"/>
            <Label x:Name="price2" FontSize="18" HorizontalOptions="End"
                   Grid.Row="1" Grid.Column="1" Text="price #2"/>

            <Label x:Name="item3" FontSize="18" Grid.Row="2" Text="item #3" />
            <Label x:Name="price3" FontSize="18" HorizontalOptions="End"
                   Grid.Row="2" Grid.Column="1" Text="price #3"/>

            <Label x:Name="item4" FontSize="18" Grid.Row="3" Text="item #4" />
            <Label x:Name="price4" FontSize="18" HorizontalOptions="End"
                   Grid.Row="3" Grid.Column="1" Text="price #4"/>

            <Label x:Name="item5" FontSize="18" Grid.Row="4" Text="item #5" />
            <Label x:Name="price5" FontSize="18" HorizontalOptions="End"
                   Grid.Row="4" Grid.Column="1" Text="price #5"/>

            <Label x:Name="item6" FontSize="18" Grid.Row="5" Text="item #6" />
            <Label x:Name="price6" FontSize="18" HorizontalOptions="End"
                   Grid.Row="5" Grid.Column="1" Text="price #6"/>

            <Label x:Name="guacamole" FontSize="18" FontAttributes="Italic" Text="guacamole"
                   Grid.Row="6" Grid.ColumnSpan="2" HorizontalOptions="End" VerticalOptions="End" />

        </Grid>
    </ScrollView>

</ContentPage>
```

If you used a different app name, you'll see a different namespace here.

The 10-pixel margin around the grid adds a little space between the Labels and the edge of the window.

The grid has six rows that are all the same height.

After you add this code to your *MainPage.xaml* file, don't forget to go to the *MainPage.xaml.cs* file and delete the OnCounterClicked method and count field.

The grid has two columns. The first column is five times wider than the second.

Each of these Label controls goes in a different cell. We gave each of them text like "item #1" or "price #3" to make it easier to see how the grid is laid out when you run the app.

Each price has its HorizontalOptions set to "End" so it gets aligned all the way to the right of the window.

This Label spans both columns in the bottom row, so it stretches across two cells. Try removing the Grid.ColumnSpan property—what happens?

The Label with the guacamole price has both its Horizontal and Vertical options set to "End" to align it to the bottom right corner of the cell.

The C# code for the main page

Here's the C# code for the main page of your Sloppy Joe app. We're about to give you an exercise to build a class called MenuItem that generates random sandwiches and prices. As soon as the page loads, it calls a method called MakeTheMenu that uses an array of MenuItem objects to fill in all of the prices, and one last MenuItem object to get the price for the guacamole.

```
namespace SloppyJoe;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        MakeTheMenu();
    }

    private void MakeTheMenu()
    {
        MenuItem[] menuItems = new MenuItem[6];

        for (int i = 0; i < 6; i++)
        {
            menuItems[i] = new MenuItem();
            menuItems[i].Generate();
        }

        price1.Text = menuItems[0].Price;
        item1.Text = menuItems[0].Description;
        price2.Text = menuItems[1].Price;
        item2.Text = menuItems[1].Description;
        price3.Text = menuItems[2].Price;
        item3.Text = menuItems[2].Description;
        price4.Text = menuItems[3].Price;
        item4.Text = menuItems[3].Description;
        price5.Text = menuItems[4].Price;
        item5.Text = menuItems[4].Description;
        price6.Text = menuItems[5].Price;
        item6.Text = menuItems[5].Description;

        MenuItem guacamoleMenuItem = new MenuItem();
        guacamoleMenuItem.Generate();
        guacamole.Text = "Add guacamole for " + guacamoleMenuItem.Price;
    }
}
```

Your namespace will be different if you chose a different name for your app.

Call the MakeTheMenu method as soon as the page loads.

This array will hold six references to MenuItem objects.

You'll use a for loop to create each MenuItem and call its Generate method. You'll write that Generate method in the next exercise.

Now that you have the MenuItem objects with sandwich descriptions and prices, you can use them to set the text in the Label controls.

Create one more MenuItem object to generate a random price for the guacamole. You won't use its Description field.

Exercise

MenuItem
Proteins Condiments Breads Description Price
Generate

Create the MenuItem class for your menu app.

Start by looking closely at the class diagram. It has five fields: three arrays to hold the various sandwich parts, a description, and a price. The array fields use **collection expressions** that let you create an array by putting comma-separated values between `[` square brackets `]`.

Add the MenuItem class to your project. Here's the code for the fields:

```
namespace SloppyJoe;

class MenuItem
{
    public string[] Proteins = [
                "Roast beef", "Salami", "Turkey",
                "Ham", "Pastrami", "Tofu"
    ];

    public string[] Condiments = [
                "yellow mustard", "brown mustard",
                "honey mustard", "mayo", "relish", "French dressing"
    ];

    public string[] Breads = [ "rye", "white", "wheat", "pumpernickel", "a roll" ];

    public string Description = "";
    public string Price = "";

    public void Generate()
    {
        // You'll fill in this method
    }
}
```

The MenuItem class has three array fields that use collection expressions to set their values, just like the array you saw in Chapter 3 to store playing cards.

The Generate method uses Random.Shared to choose random prices between 5.00 and 14.99 by creating a random decimal value out of two ints. We gave you the last line of code for the method:

`Price = price.ToString("c");`

The parameter to the ToString method is a format. In this case, the `"c"` format tells ToString to format the value with the local currency: if you're in the United States you'll see a $, in the UK you'll get a £, in the EU you'll see €, etc. If the values don't make sense in your currency, choose different random numbers!

Your job is to fill in the Generate method. It does the following:

- Picks a random protein from the Proteins array.
- Picks a random condiment from the Condiments array.
- Picks a random bread from the Breads array.
- Sets the description field like this: `protein + " with " + condiment + " on " + bread`.
- Sets the Price field to a random price that's at least 5.00 and less than 15.00. Pick a random int that's at least 5 and less than 15. Then pick a second random int that's at least 0 and less than 100. Multiply the second number by .01M to get a decimal value that's at least .00 and less than 1.00, and add it to the first value, and store it in a variable called `price`. Then set the Price field like this: `Price = price.ToString("c");`

Sharpen your pencil

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1, try multiplying it by 10. What do you get?

.....

Exercise Solution

```
public void Generate()
{
    string protein = Proteins[Random.Shared.Next(Proteins.Length)];
    string condiment = Condiments[Random.Shared.Next(Condiments.Length)];
    string bread = Breads[Random.Shared.Next(Breads.Length)];
    Description = protein + " with " + condiment + " on " + bread;

    int bucks = Random.Shared.Next(5, 15);
    int cents = Random.Shared.Next(0, 100);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1, try multiplying it by 10. What do you get?

Price = (Random.Shared.NextDouble() * 10 + 5).ToString("c");

We haven't talked about **accessibility** in this project yet. Shouldn't we add semantic properties to the controls in the menu app?

You're right! This is a great time to improve accessibility.

Sloppy Joe has a wheelchair ramp and braille versions of all of his menus, because he wants to make sure everyone has a chance to eat his budget-friendly sandwiches. So let's make sure our menu app is accessible too!

Start your operating system's screen reader and read the menu page.

Do this!

Windows Narrator

Start Windows Narrator (Ctrl+⊞+N). Narrator will scroll through the contents of any window when you hold down the Narrator key ((typically the Insert key, but you can change that in Narrator settings)) and press the left or right arrows. Navigate to your app, then navigate through all the controls and listen to what Narrator says.

macOS VoiceOver

Start VoiceOver (⌘+F5). VoiceOver will read the contents of any window when you hold down the VoiceOver activation key (^ control + ⌥ option) and press A. Navigate to your app and press VO+A (or ^ ⌥A), and listen to what VoiceOver says. Press the either ^ or ⌥ to stop reading.

Can we make the app more accessible?

When a screen reader narrates a window, it navigates from item to item, reading each item aloud and drawing a rectangle around it. What did you hear when you listened to the screen reader narrate your app? What did you see? Try having it read the menu while you have your eyes closed. Did you still understand everything that you needed to? It's pretty good! But accessibility is all about making things better for all of our users. Can we make it better?

Set the main header so the screen reader narrates it

You may have noticed that the first thing it said was "Home"—and if you watched carefully, you saw that was narrating the title bar. **Modify *AppShell.xaml* to change "Home" to "Sloppy Joe's menu"** and have the screen reader narrate the page again.

It would be great to have the narrator tell the user that they're looking at items on a menu. Let's try adding a SemanticProperties.Description to the `<Grid>` tag:

```
<Grid Margin="10"
      SemanticProperties.Description="Here are the items on the menu.">
```

Now try using the screen reader to narrate the window. It sounds fine in Windows, but if you're using macOS there's a problem: the screen reader won't read the items or prices. That's because if you set the SemanticProperties.Description on a control that has children, the screen reader can't reach those children anymore. This is important even if you're building software for Windows, because your MAUI apps are cross-platform, and you want your app to be accessible anywhere.

Try setting the item1 label's SemanticProperties.Description instead

OK, let's try something else. Remove the SemanticProperties.Description property from the `<Grid>` tag. Then try setting the SemanticProperties.Description on the first label:

```
<Label x:Name="item1" FontSize="18" Text="item #1"
       SemanticProperties.Description="Here are the items on the menu." />
```

Try using the screen reader again. ***It's still not right!*** When you have a Label, you always want the screen reader to read the contents of the label. Setting the SemanticProperties.Description causes the screen reader to read that description *instead of the label text*.

Go ahead and delete the SemanticDescription property from the item1 Label control (and also from the Grid, if you haven't done it already).

Brain Power

What do you think you'll do to make the screen reader say "Here are the items on the menu" followed by the randomly generated sandwich in the item1 Label control's Text property?

Use the SetValue method to change a control's semantic properties

Let's find a different way to make the screen reader say "Here are the items on the menu" before it reads the menu items. We'll still use the SemanticProperties.Description for the first menu item, but instead of using a XAML tag, we'll use C# to make sure it preserves the text.

Add this line of code to the end of your MainPage method:

```
public MainPage()
{
    InitializeComponent();
    MakeTheMenu();

    item1.SetValue(SemanticProperties.DescriptionProperty,
        "Here are the items on the menu. " + item1.Text);
}
```

If you type "item1." into Visual Studio, you won't see SemanticProperties in the IntelliSense pop-up. That's why you need to use the SetValue method to set it instead.

This code sets the SemanticProperties.Description property—in this case, it's setting it to the text "Here are the items on the menu" followed by the random sandwich generated by MenuItem. Try the screen reader one more time—now the page includes that text, and works on all operating systems.

Bullet Points

- The **`new` keyword** returns a reference to an object that you can store in a reference variable.
- You can have **multiple references** to the same object. You can change an object with one reference and access the results of that change with another.
- For an object to stay in the heap, it **has to be referenced**. Once the last reference to an object disappears, it eventually gets **garbage-collected** and the memory it used is reclaimed.
- Your .NET apps run in the **Common Language Runtime** (CLR), a "layer" between the OS and your program. The C# compiler builds your code into **Common Intermediate Language** (CIL), which the CLR executes.
- Declare **array variables** by putting square brackets after the type in the variable declaration (like bool[] trueFalseValues or Dog[] kennel).
- Use the **`new` keyword to create a new array**, specifying the array length in square brackets (like new bool[15] or new Dog[3]). The **`this` keyword** lets an object get a reference to itself.
- An AI chatbot can read your code and **add comments**, including XML documentation (XMLDoc) comments.
- Use the **Length method** on an array to get its length (like kennel.Length).
- Access an array value using its **index** in square brackets (like bool[3] or Dog[0]). Array indexes **start at 0**.
- null means a reference **points to nothing**. The compiler will warn you when a variable can **potentially be null**.
- Use the **string? type** to hold a string that's allowed to be null. Console.ReadLine can return null strings.
- You can use **Random.NextDouble** to create a random double value between 0 and 1. Multiply a random double to generate much larger random double values.
- Use **collection expressions** to initialize an array field by setting the field equal to a value starting with a square bracket, followed by a comma-delimited list of values, and ending with a square bracket.
- You can pass a **format parameter** to an object or value's ToString method. If you're calling a numeric type's ToString method, passing it a value of "c" formats the value as a local currency.
- Use a control's **SetValue method** to set its semantic properties in code, so the screen reader can include text that's generated when the app runs.

Unity Lab #2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**.

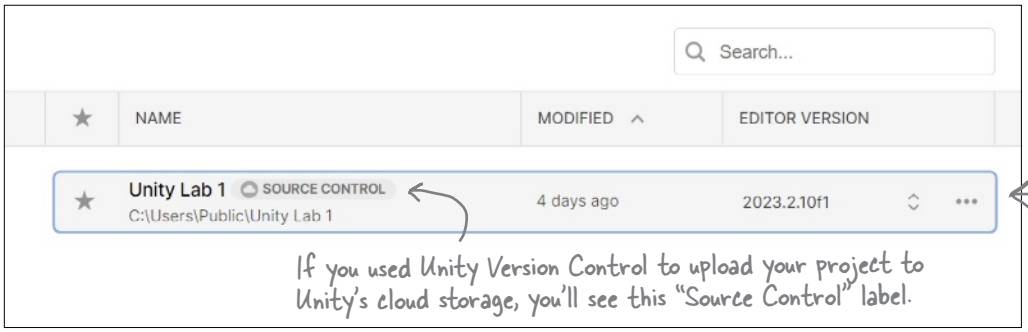
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab. You'll also start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

This Unity Lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



If you used Unity Version Control to upload your project to Unity's cloud storage, you'll see this "Source Control" label.

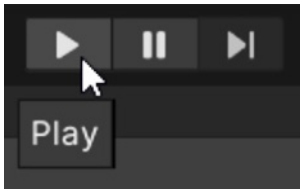
This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last lab.

Here's what you'll do in this Unity Lab:

1. **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.

2. **Use Visual Studio to edit the script.** Remember how you set the Unity editor's preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

3. **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the sphere.



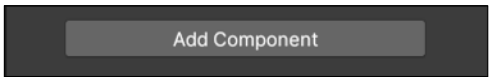
The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4. **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. Unity and Visual Studio work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

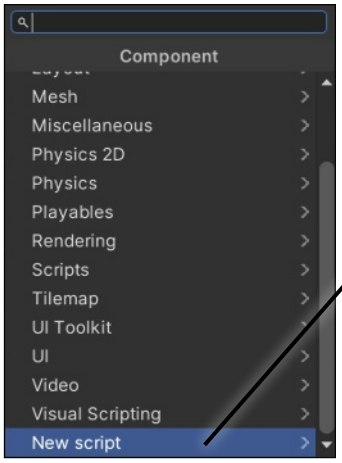
Add a C# script to your GameObject

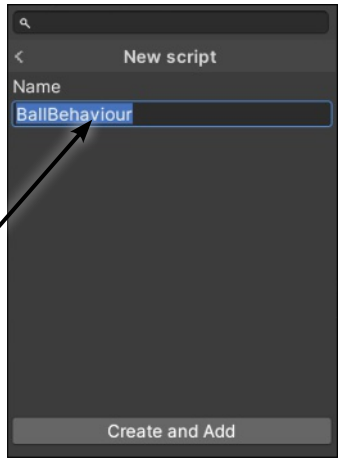
Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. It's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity ***a great tool for learning and exploring C#***.

Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are ***a lot*** of them. **Choose "New script"** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script** BallBehaviour.

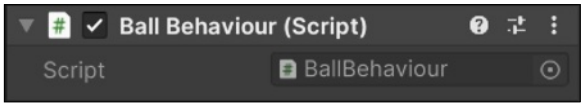




Unity code uses British spelling.

*If you're American (like us), or if you're used to the US spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

Click the "Create and Add" button to add the script. You'll see a component called *Ball Behaviour (Script)* appear in the Inspector window.



You'll also see the C# script in the Project window.

The *Project window* gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files *assets*. The Project window was displaying a folder called Assets when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called Materials appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

In the first lab, you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click your new C# script in the Assets window**. When you do, ***Unity will open your script in Visual Studio***. Your C# script contains a class called BallBehaviour with two empty methods called Start and Update:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

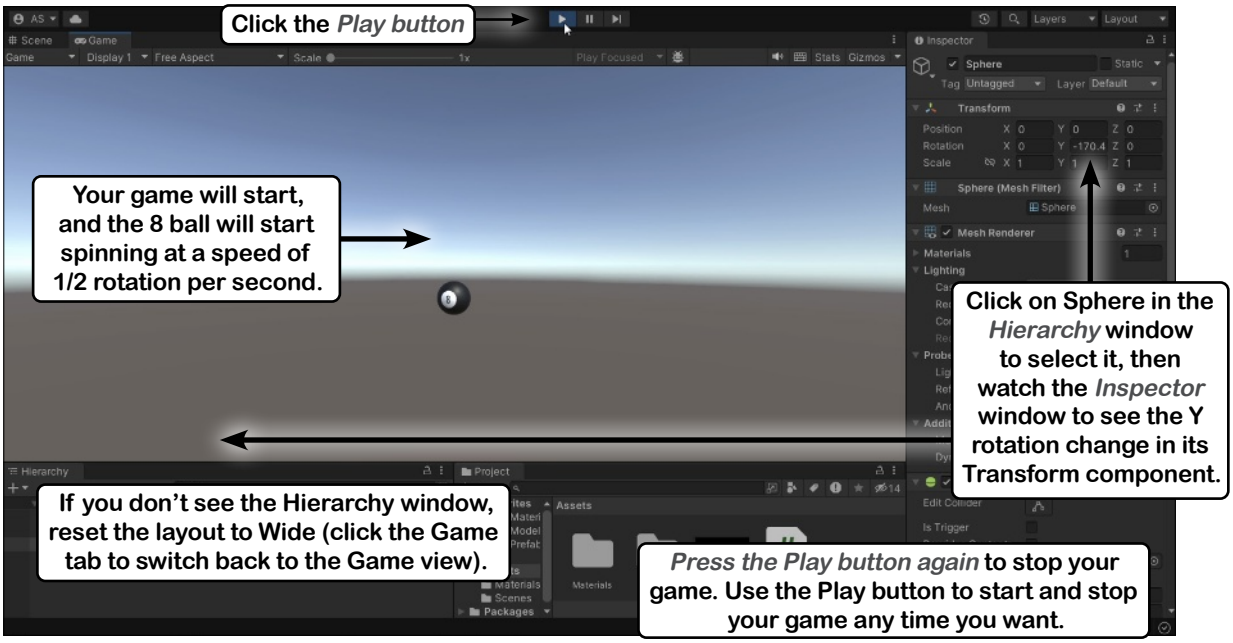
You opened your C# script in Visual Studio by clicking on it in the *Hierarchy window*, which shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

If Unity didn't launch Visual Studio and open your C# script in it, go back to the beginning of Unity Lab 1 and make sure you followed the steps to set the External Tools preferences.

Here's a line of code that will rotate your sphere. **Add it to your Update method**:

```
transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Now **go back to the Unity editor** and click the Play button in the toolbar to start your game:



Your Unity Code Up Close

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
  // Start is called before the first frame update
  void Start()
  {

  }

  // Update is called once per frame
  void Update()
  {
     transform.Rotate(Vector3.up, 180 * Time.deltaTime);
  }
}
```

You learned about namespaces in Chapter 3. When Unity created the file with the C# script, it added `using` lines so it can use code in the UnityEngine namespace and two other common namespaces.

A *frame* is a fundamental concept of animation. Unity draws one still frame, then draws the next one very quickly, and your eye interprets changes in these frames as movement. Unity calls the Update method for every GameObject before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher *frame rate*—or number of frames per second (FPS)—than a slower one.

The *transform.Rotate method* causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used Vector3.up, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the *Time.deltaTime value* comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame—it sets Time.deltaTime to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by Time.deltaTime to make sure that it rotates exactly as much as it needs to for that frame.

Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

Time.deltaTime is static—and like we saw in Chapter 3, you don't need an instance of the Time class to use it.

Add a breakpoint and debug your game

Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Visual Studio, and **add a breakpoint** on the line that you added to the Update method:

```
13      // Update is called once per frame
        Unity Message | 0 references
14      void Update()
15      {
16          transform.Rotate(Vector3.up, 180 * Time.deltaTime);
17      }
```

- In Visual Studio, click the Attach to Unity button or choose Debug >> Start Debugging (F5) from the menu.
- In VSCode, choose Run and Debug (Ctrl+Shift+D) on the left, then click Attach to Unity

Attaching to Unity **starts the debugger**. Now switch back to the Unity editor. If this is the first time you're debugging this project, the Unity editor will pop up a dialog window with these buttons:

Enable debugging for this session | Enable debugging for all projects | Cancel

Press the "Enable debugging for this session" button (or if you want to keep that pop-up from appearing again, press "Enable debugging for all projects"). Visual Studio is now ***attached*** to Unity, which means it can debug your game.

Now **press the Play button in Unity** to start your game. Since Visual Studio is attached to Unity, it ***breaks immediately*** on the breakpoint that you added, just like with any other breakpoint you've set.

Congratulations, you're now debugging a game!

Use a hit count to skip frames

Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

- In Visual Studio, right-click on the breakpoint dot (●) at the left side of the line, choose **Conditions** from the pop-up menu, select **Hit Count** and **Is a multiple of** from the dropdowns, and enter **500** in the box:

  ☑ Conditions — Hit Count | Is a multiple of | 500

- In VScode, right-click the breakpoint dot, choose **Edit breakpoint** from the dropdown, and enter **500**:

  Hit Count 500

Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. If your game is running at 60 FPS, then when you press Continue the game will run for a little over 8 seconds before it breaks again. **Press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.

VSCode lets you set a hit count for breakpoints just like Visual Studio does, but at the time we're writing this that feature does not work when debugging Unity projects.

Use the debugger to understand Time.deltaTime

You're going to be using Time.deltaTime in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with this value.

While your game is paused on the breakpoint in Visual Studio, **hover over Time.deltaTime** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for *Time.deltaTime*** by selecting Time.deltaTime and choosing Add Watch from the right-mouse menu.

```
12
13      // Update is called once per frame
        Unity Message | 0 references
14      void Update()
15      {
16          transform.Rotate(Vector3.up, 180 * Time.deltaTime);
17      }
18  }
```

Time.deltaTime 0.1975451

When you hover over Time.deltaTime, you'll see a pushpin icon next to the value. Press it to keep a small watch window pinned open in Visual Studio.

Every time the breakpoint pauses the game, your Time.deltaTime watch will show you the fraction of a second since the previous frame. Can you use this number to figure out the FPS we were getting when we took this screenshot?

Continue debugging (F5), just like with the other apps you've debugged), to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Time.deltaTime value each time it breaks, either in the pinned value or in the watch window.

Watch 1

Search (Ctrl+E) Search Depth:

Name	Value	Type
Time.deltaTime	0.1725098	float
Add item to watch		

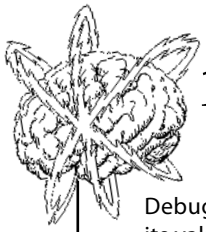
Locals | Watch 1

Press the Continue button to get another Time.deltaTime value, then another. You can get your approximate FPS by dividing 1 ÷ Time.deltaTime.

Stop debugging (Shift + F5) to stop your program. Then **start debugging again**. Since your game is still running, the breakpoint will continue to work when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and ***save it***, because the Play button doesn't automatically save the game.

The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.



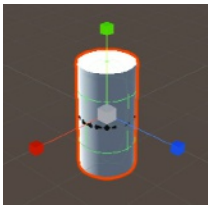
Brain Power

Debug your game again and hover over "Vector3.up" to inspect its value—you'll have to put your mouse cursor over "up." It has a value of (0.0, 1.0, 0.0). What do you think that means?

Add a cylinder to show where the Y axis is

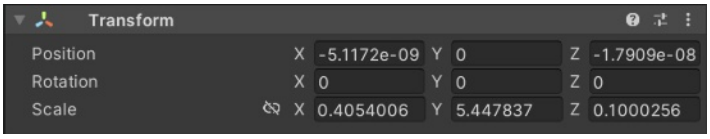
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. ***Create a new cylinder*** by choosing 3D Object >> Cylinder from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu ( ) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the toolbar: either click on it ( ) or press the R key. You should see the Scale Gizmo appear on your cylinder:



The *Scale Gizmo* looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its scale along the X and Z axes, the sphere will get uncovered.

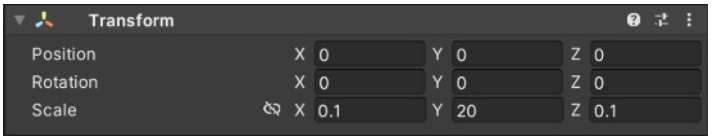
Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it toward the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z values will get much smaller.



You might notice the Position values change when you make the X and Z Scale values very small.

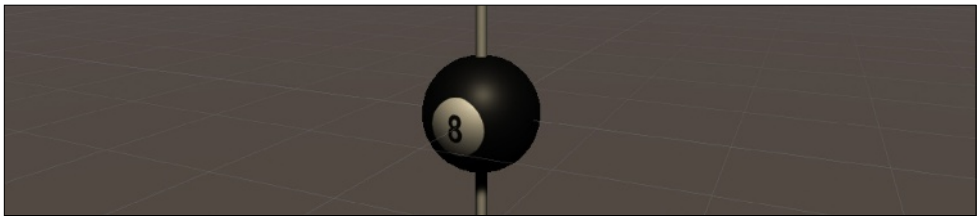
Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale as you change it. Look closely as you drag—the scale can be positive and negative. Now **reset the Transform window**.

Now **select the number inside the X box and type `0.1`**—the cylinder gets very skinny. Press Tab and type **`20`**, then press Tab again and type **`0.1`**, and press Enter.



When you edit the values in the Properties window, you can see the results update in the scene immediately.

Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



Add fields to your class for the rotation angle and speed

In Chapter 3, you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {:**

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time Update is called, it reuses the same field over and over again.

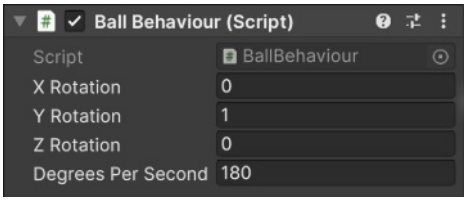
The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called `axis` and passes it to the transform.Rotate method:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
}
```

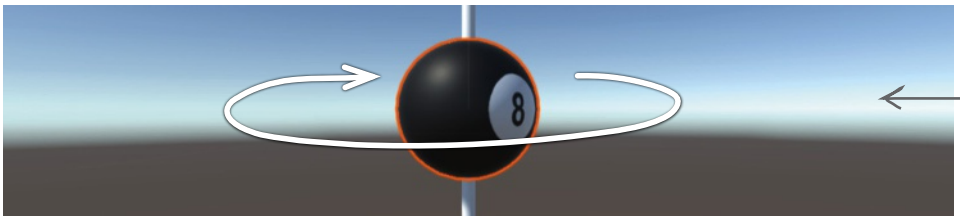
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is *not* running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. ***While it's running***, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

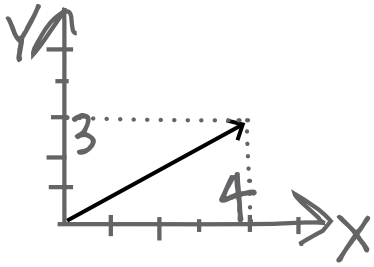
While the game is stopped, use the Unity editor to change the X Rotation field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, the ball starts rotating toward you. Make it positive again, and it starts rotating away from you.



When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand…on an intellectual level. But even those of us who took a math class that covered vectors don't always have an ***intuitive*** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really "get" how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- **It has a variable name: `axis`.**
- **It uses the `new` keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

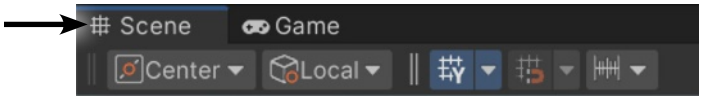
So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There's one catch: ***the ray only appears in the Scene view***. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

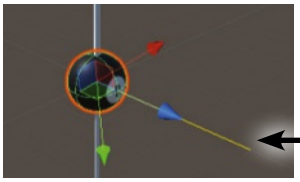
Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because Debug.DrawRay is a tool for debugging that doesn't affect gameplay at all. Use the Scene tab to **switch to the Scene view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.



Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

- Use the Inspector to **modify the BallBehaviour script's fields**. Set the X Rotation to 0, Y Rotation to 0, and **Z Rotation to 3**. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it (remember, the ray only shows up in the Scene view).



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label *in the Script component in the Inspector*. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates in the other direction.

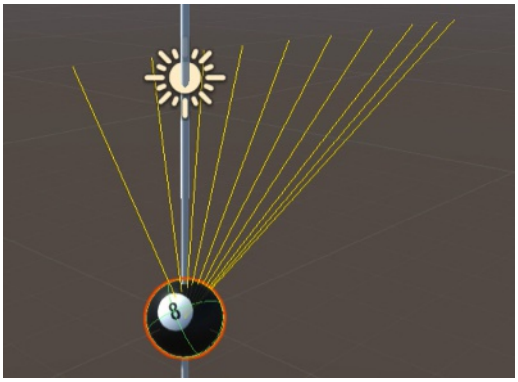
- Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

Add a duration to the ray so it leaves a trail

You can add a fourth argument to your Debug.DrawRay method call that specifies the number of seconds the ray should stay on the screen. Add **.5f** to make each ray stay on screen for half a second:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.

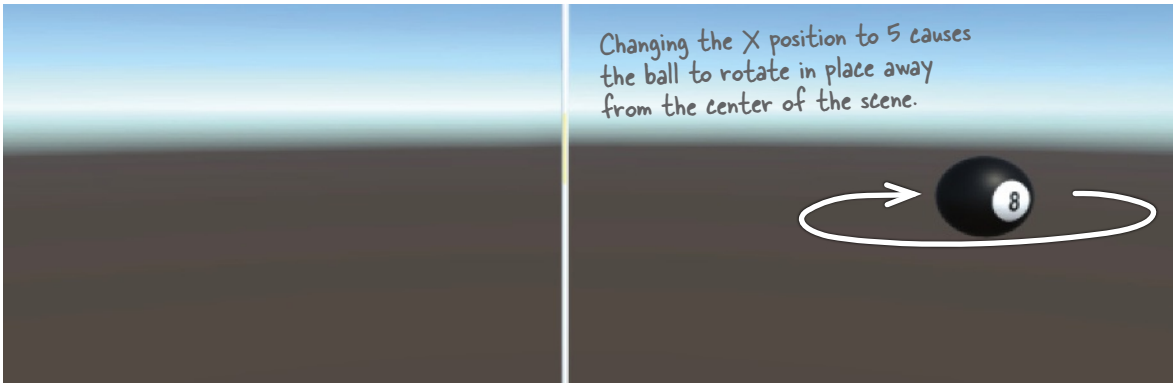


Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

You can use the Inspector window to modify the fields in a Script component while the game is running. The *field values will reset* when you stop the game. It will remember the values if you set them *while the game is stopped*.

Rotate your ball around a point in the scene

Your code calls the transform.Rotate method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu (⋮) in the BallBehaviour Script component** to reset its fields. Run the game again—now the ball will be at position (5, 0, 0) and rotating around its own Y axis.



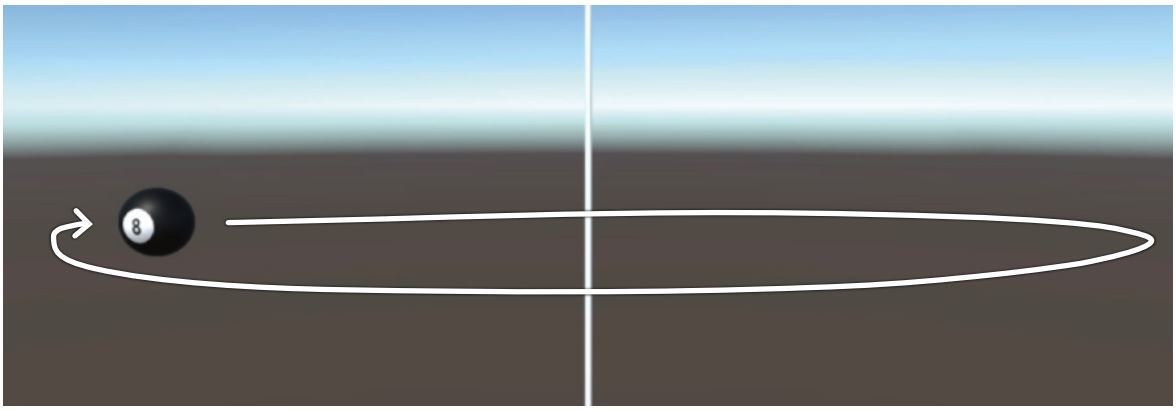
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate method you used earlier, which rotates a GameObject around its center.) Its first parameter is the point to rotate around. We'll use `Vector3.zero` for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new Update method:

This new Update method rotates the ball around the point (0, 0, 0) in the scene.

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

Now run your code. This time it rotates the ball in a big circle around the center point:

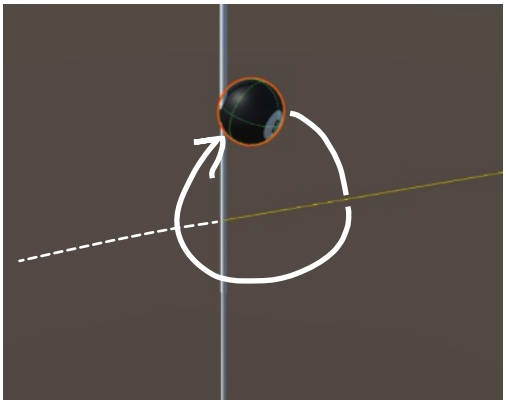


Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.
- Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.
- Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.
- Then **change the X position in** the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work by modifying properties on your GameObjects in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- Try adding a light to the scene. What happens when you use transform.rotateAround to rotate the new light around various axes?
- Here's a quick coding challenge: try using += to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by Time.deltaTime. Try adding an `if` statement that resets the field to 0 if it gets too large.

Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

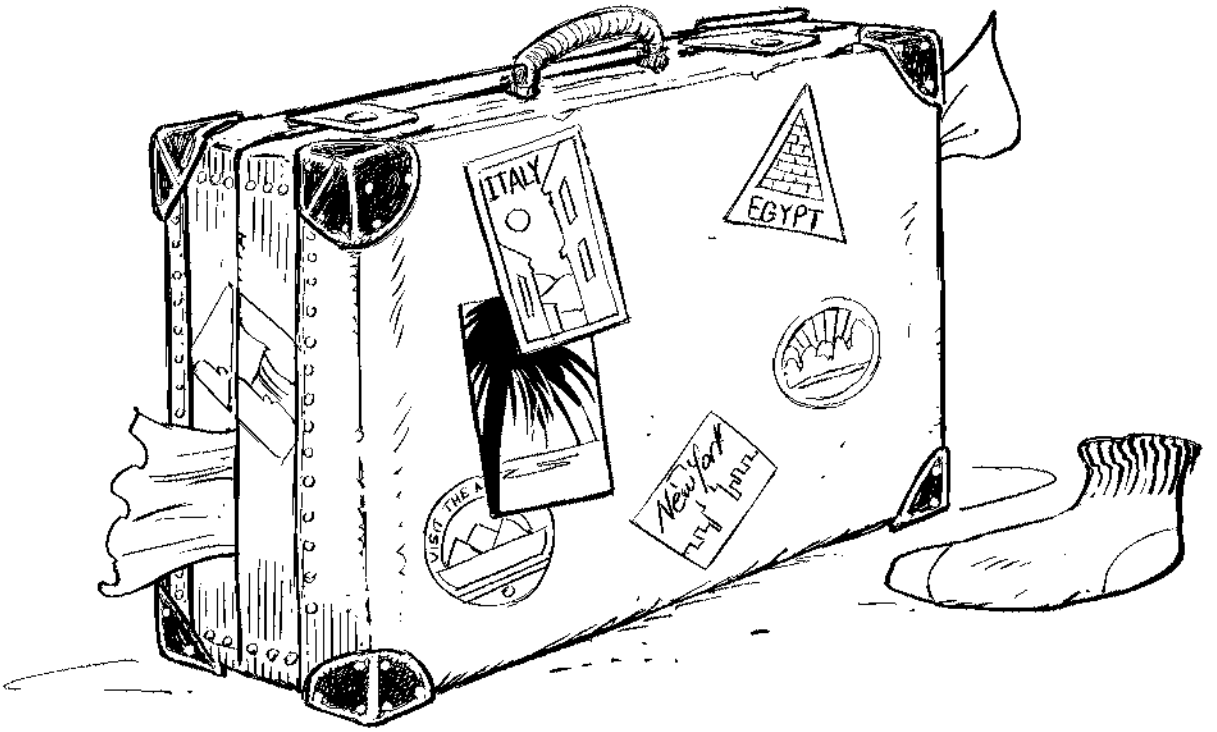
Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

Bullet Points

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's Update method will be called once per frame.
- The **transform.Rotate method** causes a GameObject to rotate a number of degrees around an axis.
- Inside your Update method, multiplying any value by **Time.deltaTime** turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes that let you modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **`new Vector3`**. (You learned about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay method** draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround method** rotates a GameObject around a point in the scene.

5 encapsulation

How objects keep their secrets



Ever wished for a little more privacy?

Sometimes your objects feel the same way. Just like you don't want anybody you don't trust reading your journal or paging through your bank statements, good objects don't let ***other*** objects go poking around their fields. In this chapter, you're going to learn about the power of **encapsulation**, a way of programming that helps you make code that's flexible, easy to use, and difficult to misuse. You'll **make your objects' data private**, and add **properties** to protect how that data is accessed—and you'll keep your object's important data from **leaking out to other objects** so they don't accidentally misuse it.

Let's help Owen roll for damage

Owen was so happy with his ability score calculator that he wanted to create more C# programs he can use for his games, and you're going to help him. In the game he's currently running, any time there's a sword attack he rolls dice and uses a formula that calculates the damage. Owen wrote down how the **sword damage formula** works in his game master notebook.

Here's the description of the sword damage formula in Owen's game master notebook.

- To find the number of hit points (HP) of damage for a sword attack, roll 3d6 (three 6-sided dice) and add "base damage" of 3HP.
- Some swords are flaming, which causes an extra 2HP of damage.
- Some swords are magic. For magic swords, the 3d6 roll is multiplied by 1.75 and rounded down, and the base damage and flaming damage are added to the result.

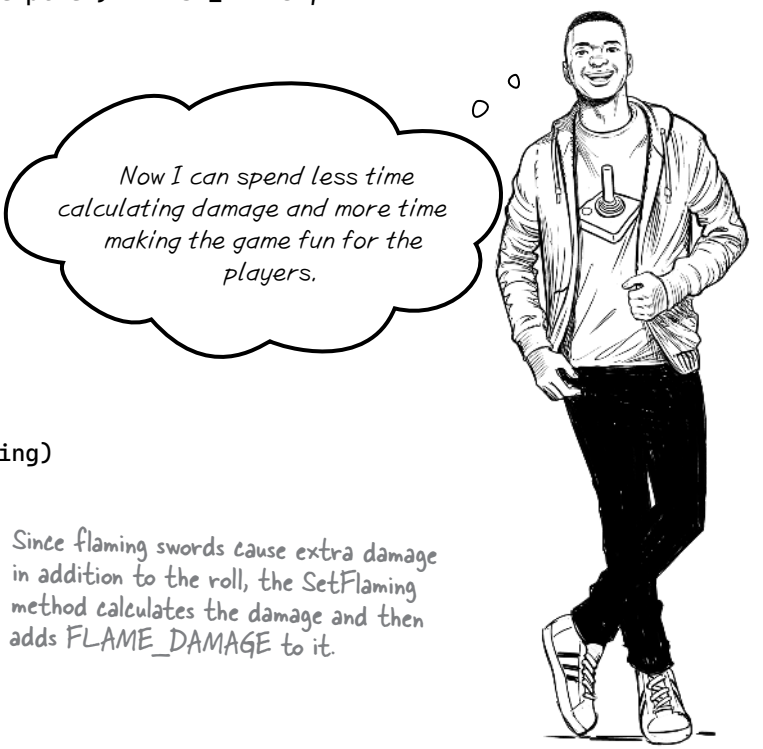
Here's **a class called SwordDamage** that does the calculation. Read through the code carefully—you're about to create an app that uses it.

```
class SwordDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public int Roll;
    public decimal MagicMultiplier = 1M;
    public int Damage;

    public void CalculateDamage()
    {
        Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE;
    }

    public void SetMagic(bool isMagic)
    {
        if (isMagic)
        {
            MagicMultiplier = 1.75M;
        }
        else
        {
            MagicMultiplier = 1M;
        }
        CalculateDamage();
    }

    public void SetFlaming(bool isFlaming)
    {
        CalculateDamage();
        if (isFlaming)
        {
            Damage += FLAME_DAMAGE;
        }
    }
}
```

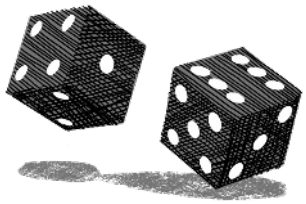
Here's a useful C# tool. Since the base damage or flame damage won't be changed by the program, you can use the `const` keyword to declare them as *constants*, which are like variables except that their value can never be changed. If you write code that tries to change a constant, you'll get a compiler error.

Here's where the damage formula gets calculated. Take a minute and read the code to see how it implements the formula.

Since flaming swords cause extra damage in addition to the roll, the SetFlaming method calculates the damage and then adds FLAME_DAMAGE to it.



Create a console app to calculate damage



Let's build a console app for Owen that uses the SwordDamage class. It will print a prompt to the console asking the user to specify whether the sword is magic and/or flaming, then it will do the calculation. Here's an example of the output of the app:

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 11 for 14 HP
```

Rolling 11 for a nonmagic, nonflaming sword will cause 11 + 3 = 14 HP of damage.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 15 for 18 HP

0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 11 for 22 HP
```

Rolling 11 for a magic sword will cause (round down 11 × 1.75 = 19) + 3 = 22

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 8 for 17 HP

0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 2
Rolled 10 for 15 HP
```

Rolling 17 for a magic flaming sword causes (round down 17 × 1.75 = 29) + 3 + 2 = 34.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 3
Rolled 17 for 34 HP
```

Exercise

Draw a class diagram for the SwordDamage class. Then **create a new console app called DamageCalculator** and add the SwordDamage class. While you're carefully entering the code, take a really close look at how the SetMagic and SetFlaming methods work, and how they work a little differently from each other. Once you're confident you understand it, you can build out the top-level statements. Here's what they'll do:

Create a new instance of the SwordDamage class.

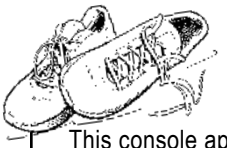
Write the prompt to the console and read the key. Call Console.ReadKey(false) so the key that the user typed is printed to the console. If the key isn't 0, 1, 2, or 3, **`return`** to end the program.

Roll 3d6 by calling Random.Shared.Next(1, 7) three times and adding the results together, and set the Roll field.

If the user pressed 1 or 3 call SetMagic(true): otherwise call SetMagic(false). You don't need an **`if`** statement to do this: `key == '1'` returns true, so you can use `||` to check the key directly inside the argument.

If the user pressed 2 or 3, call SetFlaming(true); otherwise call SetFlaming(false). Again, you can do this in a single statement using == and `||`.

Write the results to the console. Look carefully at the output and use `\n` to insert line breaks where needed.

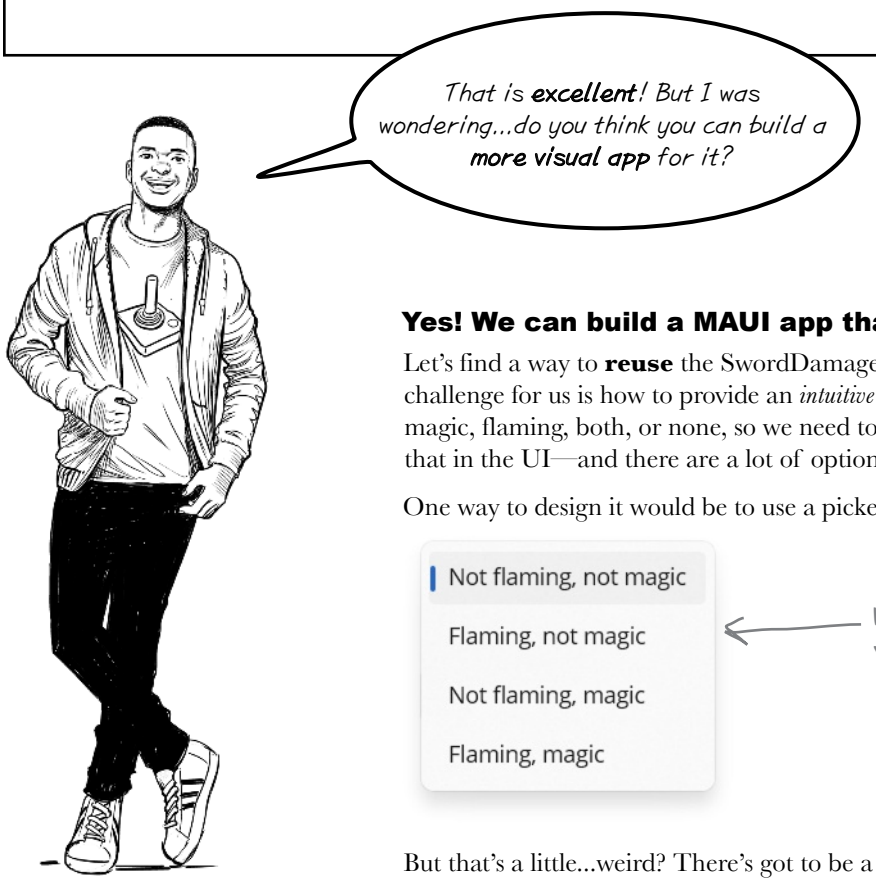


Exercise Solution

SwordDamage
Roll MagicMultiplier Damage
CalculateDamage SetMagic SetFlaming

This console app rolls for damage by creating a new instance of the SwordDamage class that we gave you and generating output that matches the example.

```
SwordDamage swordDamage = new SwordDamage();
while (true)
{
    Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                  "3 for both, anything else to quit: ");
    char key = Console.ReadKey().KeyChar;
    if (key != '0' && key != '1' && key != '2' && key != '3') return;
    int roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
        + Random.Shared.Next(1, 7);
    swordDamage.Roll = roll;
    swordDamage.SetMagic(key == '1' || key == '3');
    swordDamage.SetFlaming(key == '2' || key == '3');
    Console.WriteLine("\nRolled " + roll + " for " + swordDamage.Damage + " HP\n");
}
```



Yes! We can build a MAUI app that uses the same class.

Let's find a way to **reuse** the SwordDamage class in a MAUI app. The first challenge for us is how to provide an *intuitive* user interface. A sword can be magic, flaming, both, or none, so we need to figure out how we want to handle that in the UI—and there are a lot of options.

One way to design it would be to use a picker with four options, like this:

We think using a Picker for options would be a little weird. Do you agree?

But that's a little...weird? There's got to be a better way to design the app, right?

Design a MAUI version of the damage calculator app

Let's build a .NET MAUI damage calculator app for Owen. We'll give you the code-behind C# code for it. Your job will be to create the XAML that works with the C# code.

In this project, you'll be working with two new things that you haven't used yet:

- Your app will use two **checkboxes**. A checkbox is a control that should be very familiar to you—it's a box that displays a check when you click it, and is empty when you click it again. In MAUI, the Checkbox control has a Boolean value that's true if the box is checked and false if the box is unchecked.
- The C# code in your app will use **string interpolation** to build a string to display to the user. You've been using the + operator to build strings by concatenating values together. String interpolation does the same thing, but in a way that's easier to read.

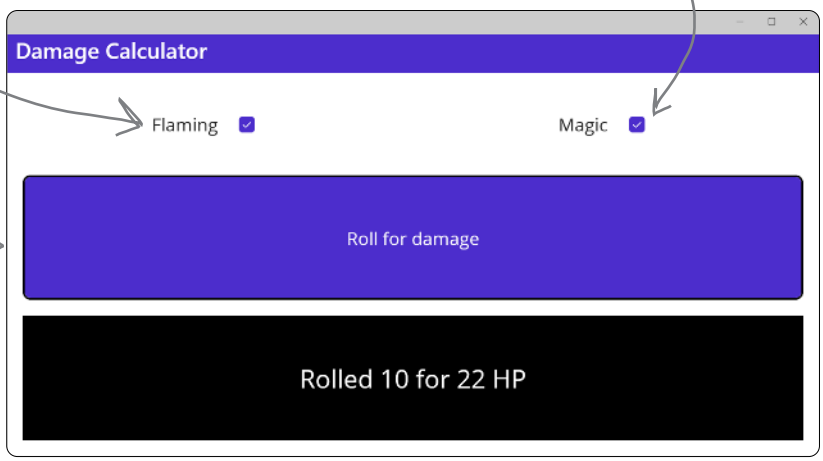
How your damage calculator app will work

Here's the main page for the damage calculator. It has two Checkbox controls to turn flaming and magic on and off, a Button to roll for damage, and a Label to display the results:

- When you click the button, it generates three random numbers to do a 3d6 roll (just like the console app did), then uses the SwordDamage class to display the damage.
- Clicking on a checkbox causes the label to update automatically. When you check or uncheck either of the checkboxes, it updates the SwordDamage fields, recalculates the damage, and updates the label.

When you check the Flaming box, it calls the SwordDamage.SetFlaming, passing it true if the box is checked and false if it's unchecked, and then calls a method to update the label to display the damage.

The Magic checkbox works just like the Flaming one, except it calls SetMagic instead of SetFlaming.



Clicking the button does a new random 3d6 roll, then updates the Roll field and displays the damage.

Exercise

It's not easy to find—don't feel bad if you don't see it yet!

There's a *bug* in the code for this app! Can you spot it?

Create a new MAUI app called **MAUIDamageCalculator**. Here's the code-behind for the *MainPage.xaml.cs* file:

```
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage();

    public MainPage()
    {
        InitializeComponent();
        swordDamage.SetFlaming(Flaming.IsChecked);
        swordDamage.SetMagic(Magic.IsChecked);
        RollDice();
    }

    private void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
            + Random.Shared.Next(1, 7);
        DisplayDamage();
    }

    private void DisplayDamage()
    {
        Damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Flaming_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.SetFlaming(e.Value);
        DisplayDamage();
    }

    private void Magic_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.SetMagic(e.Value);
        DisplayDamage();
    }

    private void Button_Clicked(object sender, EventArgs e)
    {
        RollDice();
    }
}
```

The *Checkbox control* has an IsChecked property with a Boolean value that's true if the box is checked and false if it's not checked. When the page first loads, it uses that property to set the fields on the SwordDamage object to match the checkboxes. Then it calls the RollDice method, which rolls 3d6 (just like your console app did) and calls a method to display the calculated damage.

The DisplayDamage method uses *string interpolation* to create the string and display it in a label. To use string interpolation, put a $ in front of a string, and any values to insert into the string in {curly brackets}.

The Checkbox control has a *CheckChanged event* that gets called any time it's checked or unchecked. You can get the checkbox value using `e.Value`.

Clicking the button calls the same RollDice method that the page called when it first loaded.

String interpolation

You've been using the + operator to concatenate your strings. It's a pretty powerful tool—you can use any value (as long as it's not `null`) and it will safely convert it to a string (usually by calling its ToString method). The problem is that concatenation can make your code really hard to read.

Luckily, C# gives us a great tool to concatenate strings more easily. It's called string interpolation, and to use it all you need to do is put a dollar sign in front of your string. Then to include a variable, a field, or a complex expression—or even call a method!—you put it inside curly brackets. If you want to include curly brackets in your string, just include two of them, like this: `{{ }}`

We didn't forget about accessibility! We wanted to concentrate on giving you practice laying out a grid on your own. You'll add semantic properties to the app later in the chapter. But this is a great time to add them on your own if you want to.

Exercise

Get the C# code ready for the MAUIDamageCalculator app.

- Add the SwordDamage class to your app.
- Add the code-behind C# code we gave you to the *MainPage.xaml.cs* file.
- Modify *AppShell.xaml* to change the title to "Damage Calculator."

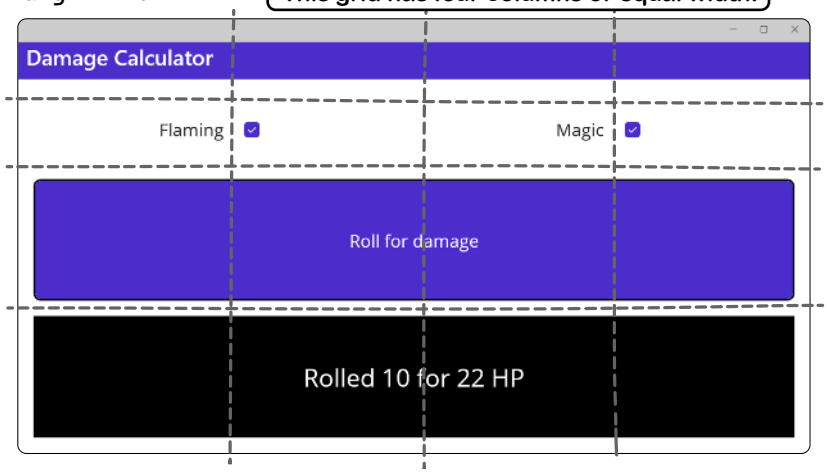
Write the XAML for the MAUIDamageCalculator app's main page.

The main page uses a grid:

We gave the Grid a margin of 20 so it has a little space around all of the controls.

```
<ContentPage ... >
    <ScrollView>
        <Grid Margin="20">
```

This grid has four columns of equal width.



This grid has three rows. The bottom two rows are twice as high as the top row.

```
        </Grid>
    </ScrollView>
</ContentPage>
```

Add two Label controls and two Checkbox controls to the top row (Grid.Row="0"), a Button control to the middle row (Grid.Row="1"), and a Label to the bottom row (Grid.Row="2") and set their properties so they match the screenshot:

- The Flaming and Magic labels have these properties: `FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"` – this causes them to be centered vertically but moved to the right of the cell.
- Give the Checkbox controls x:Name values of `"Flaming"` and `"Magic"` and have CheckChanged events call the event handler methods that we gave you. Set these properties to move them to the left of the cell: `VerticalOptions="Center" HorizontalOptions="Start" Margin="20,0,0,0"`.
- We gave the Button control a 2-pixel black border using its BorderColor and BorderWidth properties and set its font size to Medium. It spans all four columns in the row. Make it call the Clicked event handler that we gave you.
- The Label at the bottom has these properties: `x:Name="Damage" Margin="0,20,0,0" BackgroundColor="Black" TextColor="White" FontSize="Large" VerticalTextAlignment="Center" HorizontalTextAlignment="Center"`

Exercise Solution

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MAUIDamageCalculator.MainPage">

    <ScrollView>

        <Grid Margin="20">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="2*"/>
                <RowDefinition Height="2*"/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>

            <Label Text="Flaming"
                   FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

            <CheckBox x:Name="Flaming" Grid.Column="1"
                      VerticalOptions="Center" HorizontalOptions="Start"
                      CheckedChanged="Flaming_CheckedChanged" Margin="20,0,0,0" />

            <Label Text="Magic" Grid.Column="2"
                   FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

            <CheckBox x:Name="Magic" Grid.Column="3"
                      VerticalOptions="Center" HorizontalOptions="Start"
                      CheckedChanged="Magic_CheckedChanged" Margin="20,0,0,0" />

            <Button Grid.Row="1" Grid.ColumnSpan="4" Margin="0,20,0,0"
                    BorderColor="Black" BorderWidth="2"
                    Text="Roll for damage" FontSize="Medium" Clicked="Button_Clicked"/>

            <Label x:Name="Damage" Grid.Row="2" Grid.ColumnSpan="4" Margin="0,20,0,0"
                BackgroundColor="Black" TextColor="White" FontSize="Large"
                VerticalTextAlignment="Center" HorizontalTextAlignment="Center" />
        </Grid>

    </ScrollView>

</ContentPage>
```

The bottom two rows are twice as tall as the top row.

The four columns are all the same width.

We asked you to set the HorizontalOptions properties for these Label and Checkbox controls so the labels are at the right sides of their cells and the checkboxes are at the left sides so they're close to each other.

The Checkbox controls call the CheckChanged event handlers in the C# code-behind that we gave you.

We gave you many of the properties to help you make your app match our screenshot. But this is a great chance to *get creative*! Try different colors, font sizes, horizontal and vertical options, and even grid layouts. Experimenting is a great way to get these ideas into your brain quickly.

Tabletop talk (or maybe...dice discussion?)

It's game night! Owen's entire gaming party is over, and he's about to unveil his brand-new sword damage calculator. Let's see how that goes.

We told you there was a bug in the code! How do you think you'll fix it?

OK, Party, we've got a new table rule. Prepare to be dazzled by this stunning new feat of **technological amazement**.

Jayden: Owen, what are you talking about?

Owen: I'm talking about this new app that will calculate sword damage…***automatically***.

Matthew: Because rolling dice is so very, very hard.

Jayden: Come on, people, no need for sarcasm. Let's give it a chance.

Owen: Thank you, Jayden. This is a perfect time too because Brittany just attacked the rampaging were-cow with her flaming magic sword. Go ahead, B. Give it a shot.

Brittany: OK. We just started the app. I checked the Magic box. Looks like it's got an old roll in there, let me click roll to do it again, and…

Jayden: Wait, that's not right. Now you rolled 14, but ***it still says 3 HP***. Click it again. Rolled 11 for 3 HP. Click it some more. 9, 10, 5, all give 3 HP. Owen, what's the deal?

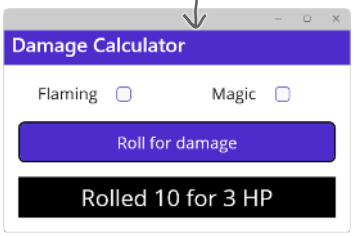
Brittany: Hey, ***it sort of works***. If you click Roll, then check the boxes a few times, eventually it gives the right answer. Looks like I rolled 10 for 22 HP.

Jayden: You're right. We just have to click things in a **really specific order**. *First* we click Roll, *then* we check the right boxes, and *just to be sure* we check the Flaming box twice.

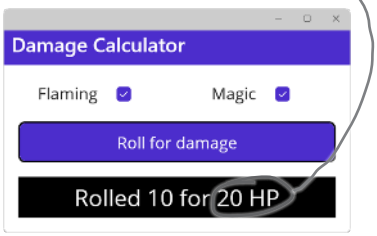
Owen: You're right. If we do things in **exactly that order**, the program works. But if we do it in any other order, it breaks. OK, we can work with this.

Matthew: Or…maybe we can just do things the normal way, with real dice?

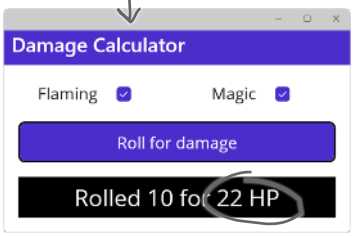
Brittany and Jayden are right. The program works, but only if you do things in a specific order. When it starts up, something's wrong—it always calculates 3HP.



Let's try to calculate damage for a flaming magic sword by checking Flaming first, then Magic second. Uh-oh—that number is wrong.



But once we click the Flaming box twice, it displays the right number.



Let's try to fix that bug

When you run the program, what's the first thing that it does? Let's take a closer look at this method at the very top of the MainPage class in the code-behind for the page in *MainPage.xaml.cs*:

```
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage();

    public MainPage()
    {
        InitializeComponent();
        swordDamage.SetMagic(Flaming.IsChecked);
        swordDamage.SetFlaming(Magic.IsChecked);
        RollDice();
    }
```

This method is a *constructor*. It *doesn't have a return type* and its name *matches the class name*. Your MAUI page is actually an instance of a class called MainPage. It gets called when the MainPage class is first instantiated, so we can use it to initialize the instance.

You've been using this method since Chapter 1. Now you know its name: it's a constructor.

When a class has a **constructor**, it's the very first thing that gets run when a new instance of that class is created. When your app starts up and creates an instance of MainPage, first it initializes the fields—including creating a new SwordDamage object—and then it calls the constructor.

The program calls RollDice just before showing you the window, and we see the problem every time we click Roll, so ***maybe*** we can fix this by guessing what's wrong, and trying out a possible solution to the problem?

Make these changes to the RollDice method so it sets the SwordDamage fields just like the constructor does:

```
public void RollDice()
{
    swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
        + Random.Shared.Next(1, 7);
    swordDamage.SetFlaming(Flaming.IsChecked);
    swordDamage.SetMagic(Magic.IsChecked);
    DisplayDamage();
}
```

Redo this!

Maybe we just need to call the SwordDamage object's methods before we display the damage. Will that fix the bug?

Now **test your code**. Run your program and click the button a few times. So far, so good—the numbers look correct. Now **check the Magic box** and click the button a few more times. OK, it looks like our fix worked! There's just one more thing to test. **Check the Flaming box** and click the button and…

Oops! It's still not working

It's still not working. When you click the button, it does the 1.75 magic multiplier, but it doesn't add the extra 3 HP for flaming. You still need to check and uncheck the Flaming checkbox to get the right number. So the app is still broken.

*We took a guess and **quickly** wrote some code, But it didn't fix the problem because **we didn't really think** about what actually caused the bug.*

Use Debug.WriteLine to print diagnostic information

In the last few chapters, you used the debugger to track down bugs, but that's not the only way developers find problems in their code. In fact, when professional developers are trying to track down bugs in their code, one of the most common things they'll do first is to **add statements that print lines of output**, and that's exactly what we'll do to track down this bug.

Open the Output window in Visual Studio by choosing **Output** (Ctrl+W+O or Ctrl+Alt+O) from the View menu in Visual Studio, or **Debug Console** from the View menu in VSCode.

Luckily, .NET gives us a really useful method for this: **Debug.WriteLine**. Any time you want to print output lines just for debugging purposes, call that method. The Debug class is in the System.Diagnostics namespace, so start by adding a `using` line to the top of your SwordDamage class file:

```
using System.Diagnostics;
```

Next, **add a Debug.WriteLine statement** to the end of the CalculateDamage method:

```
public void CalculateDamage()
{
    Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE;
    Debug.WriteLine($"CalculateDamage set Damage to {Damage} (roll: {Roll})");
}
```

Now add another Debug.WriteLine statement to the end of the SetMagic method, and one more to the end of the SetFlaming method. They should be identical to the one in CalculateDamage, except that they print "SetMagic" or "SetFlaming" instead of "CalculateDamage" to the output:

```
public void SetMagic(bool isMagic)
{
    // the rest of the SetMagic method stays the same
    Debug.WriteLine($"SetMagic finished: Damage = {Damage} (roll: {Roll})");
}

public void SetFlaming(bool isFlaming)
{
    // the rest of the SetFlaming method stays the same
    Debug.WriteLine($"SetFlaming finished: Damage = {Damage} (roll: {Roll})");
}
```

Run your app again. Now you'll see debug messages in the application output window.

Check the Output window in Visual Studio or Debug Console in VSCode to see the debug messages.

Always *think* about what caused a bug before you try to fix it.

When something goes wrong in your code, it's ***really tempting to jump right in*** and immediately start writing more code to try to fix it. It may feel like you're taking action quickly, but it's way too easy to just add more buggy code. You'll get better results if you take the time to ***reproduce the bug***: find a way to replicate the bug so you can get clues about it. It's always safer to take the time to figure out what really caused the bug, rather than just try to stick in a quick fix. Diagnostic tools like Debug.WriteLine are there to help you think through the problems.

You can sleuth out this bug without setting any breakpoints. Developers do this all the time. Now you can do it too!

Sleuth it Out

A Scandal at the Gaming Table

Let's use the **Output window** to debug the app. You can always press the Clear All (or) button to clear it.

Run your program and watch the output. As it loads, you'll see a bunch of lines with messages about the CLR loading various DLLs (that's normal, just ignore them for now). Then you'll see this output:

```
CalculateDamage set Damage to 3 (roll: 0)
SetMagic finished: Damage = 3 (roll: 0)
```

These two lines will be printed twice. Can you figure out why?

That gives us a clue about the first mystery—why does the app always show 3 HP when it starts up, no matter what the roll is? It's because it's using 0 for the roll. ***It must not be setting the SwordDamage.Roll property.***

Let's keep looking for more clues. Check the Flaming box. When we did it, our roll was 14, so this is what it printed:

```
CalculateDamage set Damage to 17 (roll: 14)
SetFlaming finished: Damage = 19 (roll: 14)
```

Make sure you modified RollDice to call SetFlaming and SetMagic, or you won't see this output.

19 is the correct answer for that roll: 14 plus base damage of 3 plus 2 for a flaming sword. So far, so good.

Thanks to the debug messages, you can see what methods are being called. The SetFlaming method called CalculateDamage, which calculated 17 and then executed the first Debug.WriteLine statement. It then added FLAME_DAMAGE (2) to get the value up to 19, and finally executed the second Debug.WriteLine statement.

Now press the button to roll again. The program should write four more lines to the Output window:

```
CalculateDamage set Damage to 15 (roll: 12)
SetFlaming finished: Damage = 17 (roll: 12)
CalculateDamage set Damage to 15 (roll: 12)
SetMagic finished: Damage = 15 (roll: 12)
```

Wait a minute! SetFlaming added 2 to Damage to set it to 17. But then SetMagic called CalculateDamage, which set it back to 15.

We rolled a 12, so it should calculate 17 HP. But it didn't—it calculated 15. What does the debug output tell us?

First the RollDice method called SetFlaming, which calculated 17 just like before.

But ***then*** it called the CalculateDamage method, which **overwrote the Damage field** and set it back to 15.

The problem is that **SetFlaming was called before CalculateDamage**, so even though it added the flame damage correctly, calling CalculateDamage afterward undid that. So the real reason that the program doesn't work is that the fields and methods in the SwordDamage class ***need to be used in a very specific order***:

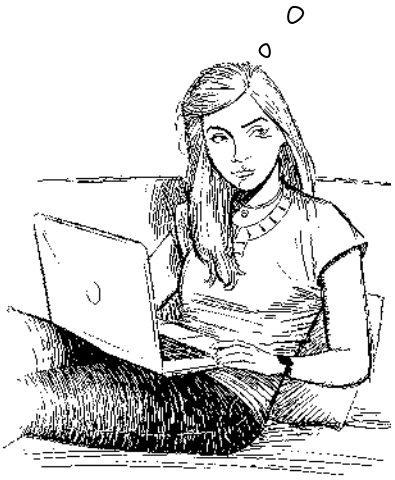
1. Set the Roll field to the 3d6 roll.
2. Call the SetMagic method.
3. Call the SetFlaming method.
4. Do not call the CalculateDamage method. SetFlaming does that for you.

Debug.WriteLine is one of the most basic (and useful!) debugging tools in your developer toolbox. Sometimes the quickest way to sleuth out a bug in your code is to strategically add Debug.WriteLine statements to give you important clues that help you crack the case.

And that's why the console app worked, but the MAUI version didn't. The console app used the SwordDamage class in the specific way that it works. The MAUI app ***called them in the wrong order***, so it got incorrect results.

Now we know what caused the bug! But we won't fix this project yet. We'll return to fix the project after learning more about encapsulation.

So the methods just need to be **called in a particular order**. What's the big deal? I just need to flip around the order that I call them, and my code will start working.



People won't always use your classes in exactly the way you expect.

And most of the time those "people" who are using your classes are you! You might be writing a class today that you'll be using tomorrow, or next month. Luckily, C# gives you a powerful technique to make sure your classes are always used correctly—even when people do things you never thought of. It's called **encapsulation** and it's really helpful for working with objects. The goal of encapsulation is to restrict access to the "guts" of your classes so that all of the class members are ***safe to use and difficult to misuse***. This lets you design classes that are much more difficult to use incorrectly—and that's a ***great way to prevent bugs*** like the one you sleuthed out in your sword damage calculator.

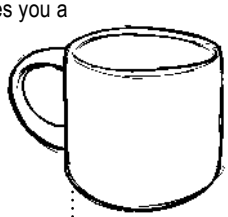
there are no Dumb Questions

Q: What's the difference between Console.WriteLine and Debug.WriteLine?

A: The Console class is used by console apps to get input from and send output to the user. It uses the three **standard streams** provided by your operating system: standard input (stdin), standard output (stdout), and standard error (stderr). Standard input is the text that goes into the program, and standard output is what it prints. (If you've ever piped input or output in a shell or command prompt using <, >, |, <<, >>, or ||, you've used stdin and stdout.) The Debug class is in the System.Diagnostics namespace, which gives you a hint about its use: it's for helping diagnose problems by tracking down and fixing them. Debug.WriteLine sends its output to **trace listeners**, or special classes that monitor diagnostic output from your program and write them to the console, log files, or a diagnostic tool that collects data from your program for analysis.

Q: Can I use constructors in my own code?

A: Absolutely. A constructor is a method that the CLR calls when it first creates a new instance of an object. It's just an ordinary method—there's nothing weird or special about it. You can add a constructor to any class by declaring a method **without a return type** (so no void, int, or other type at the beginning) that has the **same name as the class**. Any time the CLR sees a method like that in a class, it recognizes it as a constructor and calls it any time it creates a new object and puts it on the heap.



Relax

We're not done with constructors.

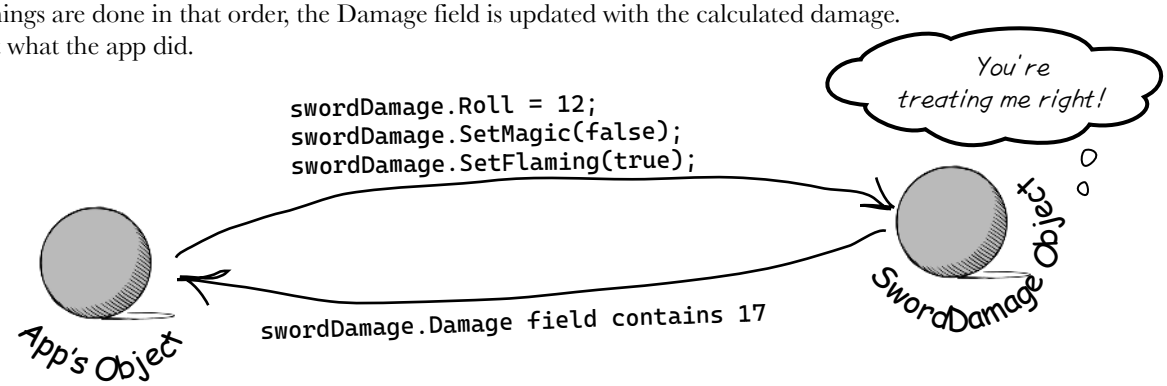
We'll do a lot more work with constructors later in the chapter. For now, just think of a constructor as a special method that you can use to initialize an object.

It's easy to accidentally misuse your objects

Owen's app ran into problems because we assumed that the CalculateDamage method would, well, calculate the damage. It turned out that **it wasn't safe to call that method directly** because it replaced the Damage value and erased any calculations that were already done. Instead, we needed to let the SetFlaming method call CalculateDamage for us—but **even that wasn't enough**, because we *also* had to make sure that SetMagic was always called first. So even though the SwordDamage class *technically* works, it causes problems when code calls it in an unexpected way.

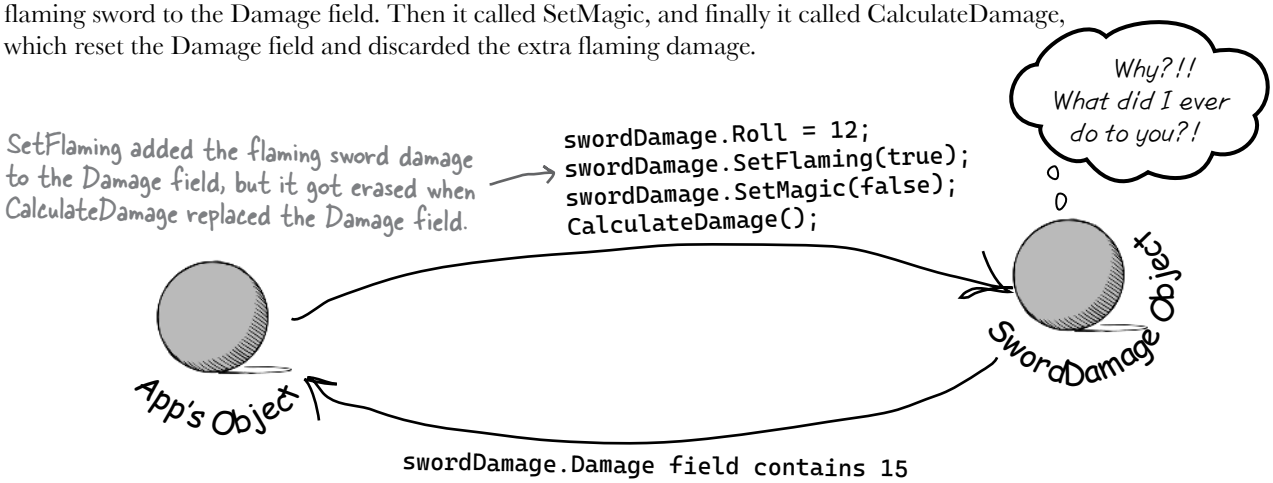
How the SwordDamage class expected to be used

The SwordDamage class gave the app a good method to calculate the total damage for a sword. All it had to do was set the roll, then call the SetMagic method, and finally call the SetFlaming method. If things are done in that order, the Damage field is updated with the calculated damage. But that's not what the app did.



How the SwordDamage class was actually used

Instead, it set the Roll field, then it called SetFlaming, which added the extra damage for the flaming sword to the Damage field. Then it called SetMagic, and finally it called CalculateDamage, which reset the Damage field and discarded the extra flaming damage.



Encapsulation means keeping some data in a class private

There's a way to avoid the problem of misusing your objects: make sure that there's only one way to use your class. C# helps you do that by letting you declare some of your fields as **private**. So far, you've only seen public fields. If you've got an object with a public field, any other object can read or change that field. If you make it a private field, then **that field can only be accessed from inside that object** (or by another instance of the same class).

When in doubt, make it private

Worried about trying to figure out which fields and methods to make private? Start by making every member private, and change them to public only if you need to. **In this case, laziness can work to your advantage.** If you leave off the "private" or "public" declaration, then ***C# will just assume that your field or method is private***.

```
class SwordDamage
{
   public const int BASE_DAMAGE = 3;
   public const int FLAME_DAMAGE = 2;

   public int Roll;
   private decimal magicMultiplier = 1M;
   public int Damage;

   private void CalculateDamage()
   {
      ...
```

If you want to make a field private, all you need to do is use the `private` keyword when you declare it. This tells C# that if you've got an instance of SwordDamage, its magicMultiplier field can *only be read and written* by methods in an instance of SwordDamage. Other objects won't see them at all.

Did you notice that we also changed the private field name so it starts with a lowercase letter?

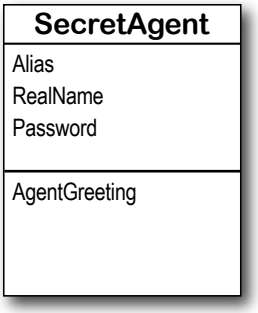
By making the CalculateDamage method private, we prevent the app from accidentally calling it and resetting the Damage field. Changing the fields involved in the calculation to make them private keeps an app from interfering with the calculation. When you make some data private and then write code to use that data, it's called encapsulation. When a class protects its data and provides members that are safe to use and difficult to misuse, we say that it's *well-encapsulated*.

en-cap-su-la-ted, adj.
enclosed by a protective coating or membrane. *The divers were fully* ***encapsulated*** *by their submersible, and could only enter and exit through the airlock.*

Use encapsulation to control access to your class's methods and fields

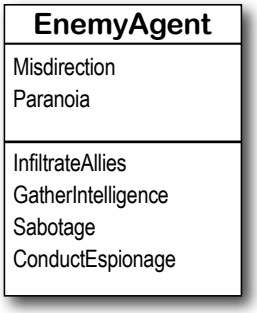
When you make all of your fields and methods public, any other class can *access* them. Everything your class does and knows about becomes an open book for every other class in your program…and you just saw how that can cause your program to behave in ways you never expected.

That's why the `public` and `private` keywords are called **access modifiers**: they modify access to class members. Encapsulation lets you control what you share and what you keep private inside your class. Let's see how this works.



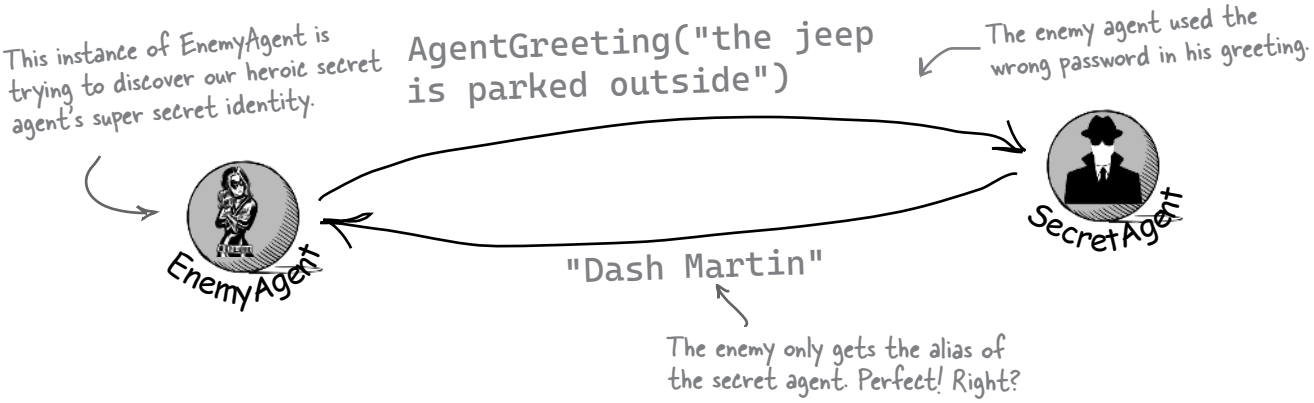
1. Super-spy Herb Jones is a ***secret agent object in a spy*** defending life, liberty, and the pursuit of happiness as a secret agent undercover in enemy territory. His object is an instance of the SecretAgent class.

```
RealName: "Herb Jones"
Alias: "Dash Martin"
Password: "the crow flies at midnight"
```



2. Agent Jones has a plan to help him evade the enemy agent object. He added an AgentGreeting method that takes a password as its parameter. If he doesn't get the right password, he'll only reveal his alias, Dash Martin.

3. Seems like a foolproof way to protect the agent's identity, right? As long as the agent object that calls it doesn't have the right password, the agent's name is safe.



But is the RealName field REALLY protected?

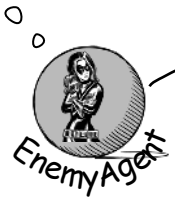
So as long as the enemy doesn't know any SecretAgent object passwords, the agents' real names are safe. Right? But that doesn't do any good if that data's kept in public fields.

```
public string RealName;
public string Password;
```

Making your fields public means they can be accessed (and even changed!) by any other object.

Aha! He left the field public! Why go through all of the trouble to guess the password for the AgentGreeting method? I can just get his name directly!

```
string iSpy = herbJones.RealName;
```



What can Agent Jones do? He can use **private** fields to keep his identity secret from enemy spy objects. Once he declares the realName field as private, the only way to get to it is ***by calling methods that have access to the private parts of the class***. So the enemy agent is foiled!

The EnemyAgent object can't access the SecretAgent's private fields because they're instances of different classes.

Just replace `public` with `private`, and now the field is hidden from any object that isn't an instance of the same class. Keeping the right fields and methods private makes sure no outside code is going to change values you're using when you don't expect it. We renamed the fields to *start with lowercase letters* to make our code more readable.

```
private string realName;
private string password;
```

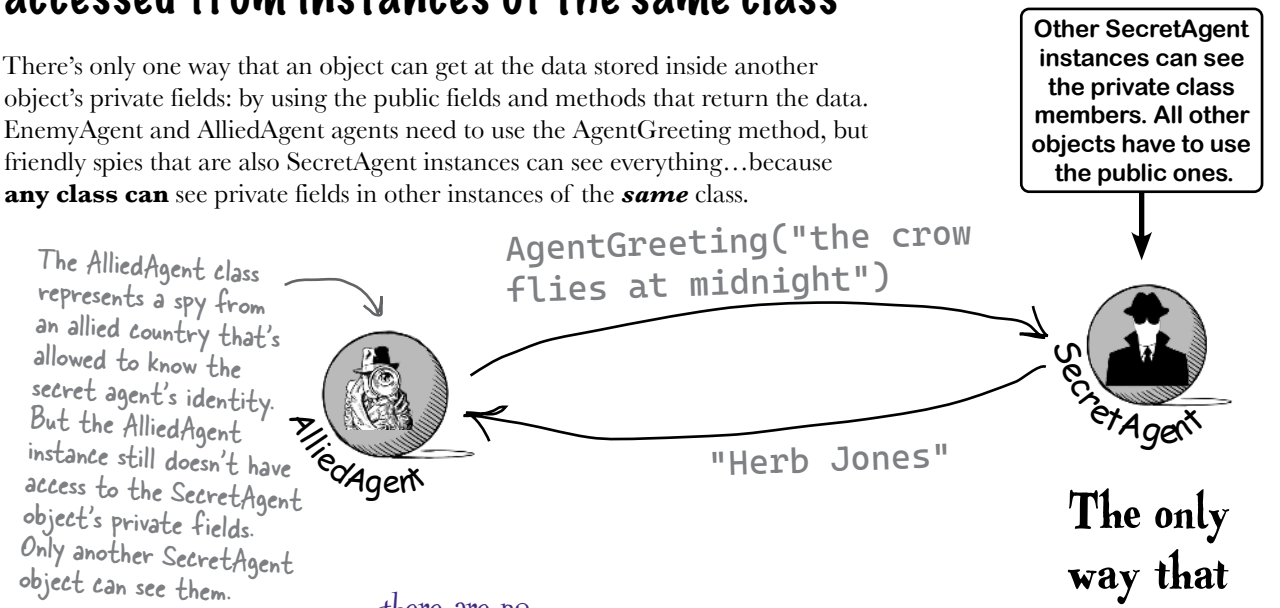
When you start public fields with uppercase letters and private fields with lowercase ones, you'll know which fields are public or private when you see them used—without having to go back to the declaration.

Brain Power

Making the methods and fields in the damage calculator app private prevents bugs by keeping the app from using them directly. But there's still a problem! We'll still get the wrong answer if SetMagic is called before SetFlaming. Can the `private` keyword help you prevent that?

Private fields and methods can only be accessed from instances of the same class

There's only one way that an object can get at the data stored inside another object's private fields: by using the public fields and methods that return the data. EnemyAgent and AlliedAgent agents need to use the AgentGreeting method, but friendly spies that are also SecretAgent instances can see everything…because **any class can** see private fields in other instances of the ***same*** class.



The only way that one object can get to data stored in a private field inside another object of a <u>different</u> class is by using public methods that return the data.

there are no Dumb Questions

Q: Why would I ever want a field in an object that another object can't read or write?

A: Sometimes a class needs to keep track of information that is necessary for it to operate, but that no other object really needs to see—and you already saw an example of this. In Chapter 4, you saw that the Random class used special *seed values* to initialize the pseudo-random number generator. Under the hood, it turns out that every instance of the Random class actually contains an array of several dozen numbers that it uses to make sure that the Next method always gives you a random number. But that array is private—when you create an instance of Random, you can't access that array. If you had access to it, you might be able to put values in it that would cause it to give nonrandom values. So the seeds have been completely encapsulated from you.

Q: OK, so I need to access private data through public methods. What happens if the class with the private field doesn't give me a way to get at that data, but my object needs to use it?

A: Then you can't access the data from outside the object. When you're writing a class, you should always make sure that you give other objects some way to get at the data they need. Private fields are a very important part of encapsulation, but they're only part of the story. Writing a class with good encapsulation means giving a sensible, easy-to-use way for other objects to get the data they need, without giving them access to hijack data your class depends on.

Q: Hey, I just noticed that when I use "Generate method" in the IDE, it uses the `private` keyword. Why does it do that?

A: Because it's the safest thing for the IDE to do. Not only are the methods created with "Generate method" private, but when you double-click on a control to add an event handler, the IDE creates a private method for that too. The reason is that it's **safest to make a field or method private** to prevent the kinds of bugs that we saw in the damage calculator. You can always make your class members public later if you need another class to access the data.

Exercise

Let's get a little practice using the **`private`** keyword by **creating a small Hi-Lo game**. The game starts with a pot of 10 bucks, and it picks a random number from 1 to 10. The player will guess if the next number will be higher or lower. If the player guesses right they win a buck, otherwise they lose a buck. Then the next number becomes the current number, and the game continues.

Go ahead and **create a new console app** for the game. Here are the top-level statements:

```
Console.WriteLine("Welcome to HiLo.");
Console.WriteLine($"Guess numbers between 1 and {HiLoGame.MAXIMUM}.");
HiLoGame.Hint();
while (HiLoGame.GetPot() > 0)
{
    Console.WriteLine("Press h for higher, l for lower, ? to buy a hint,");
    Console.WriteLine($"or any other key to quit with {HiLoGame.GetPot()}.");
    char key = Console.ReadKey(true).KeyChar;
    if (key == 'h') HiLoGame.Guess(true);
    else if (key == 'l') HiLoGame.Guess(false);
    else if (key == '?') HiLoGame.Hint();
    else return;
}
Console.WriteLine("The pot is empty. Bye!");
```

Put the HiLoGame class in the global namespace we learned about in Chapter 3 by leaving out the namespace declaration. Then you don't need to add a using statement in your top-level statements.

Next, add a **static class** called HiLoGame and **add the following members**. Since this is a static class, all of the members need to be static. Make sure to include either **`public`** or **`private`** in the declaration for each member:

1. A constant integer **MAXIMUM** that defaults to 10. Remember, you can't use the **`static`** keyword with constants.
2. Two int fields called **currentNumber** and **nextNumber** that are both initialized to random numbers.
3. An integer field called **pot** with the number of bucks in the pot. **Make this field `private`.**

We made pot private because we don't want other classes to be able to add money, but the top-level statements still need to be able to print the size of the pot to the console. Look carefully at the top-level statements—can you figure out how the top-level statements get the value of the pot field without making the field writeable?

4. A **method** called **`Guess`** with a bool parameter called **`higher`** that does the following (look closely at the top-level statements to see how it's called):
   - If the player guessed higher and the next number is >= the current number **OR** if the player guessed lower and the next number is <= the current number, **write** "You guessed right!" to the console and **increment** the value in the pot.
   - Otherwise, it **writes** "Bad luck, you guessed wrong." to the console and decrements the pot.
   - It sets currentNumber to nextNumber, then sets nextNumber to a new random number for the player to guess.
   - It writes "The current number is {currentNumber}" to the console.
5. A **Hint** method that finds half the maximum and writes either "The current number is {currentNumber}, the next is at least {half}" or "The current number is {currentNumber}, the next is at most {half}" then decrements the pot.

Exercise Solution

Here's the rest of the code for the Hi-Lo game. The game starts with a pot of 10 bucks, and it picks a random number from 1 to 10. The player will guess if the next number will be higher or lower. If the player guesses right they win a buck, otherwise they lose a buck. Then the next number becomes the current number, and the game continues.

Here's the code for the HiLoGame class:

If you try to add the `static` keyword to a constant, you'll get a compiler error because *all constants are static*. Try adding one to any class—you can access it from another class just like any other static field.

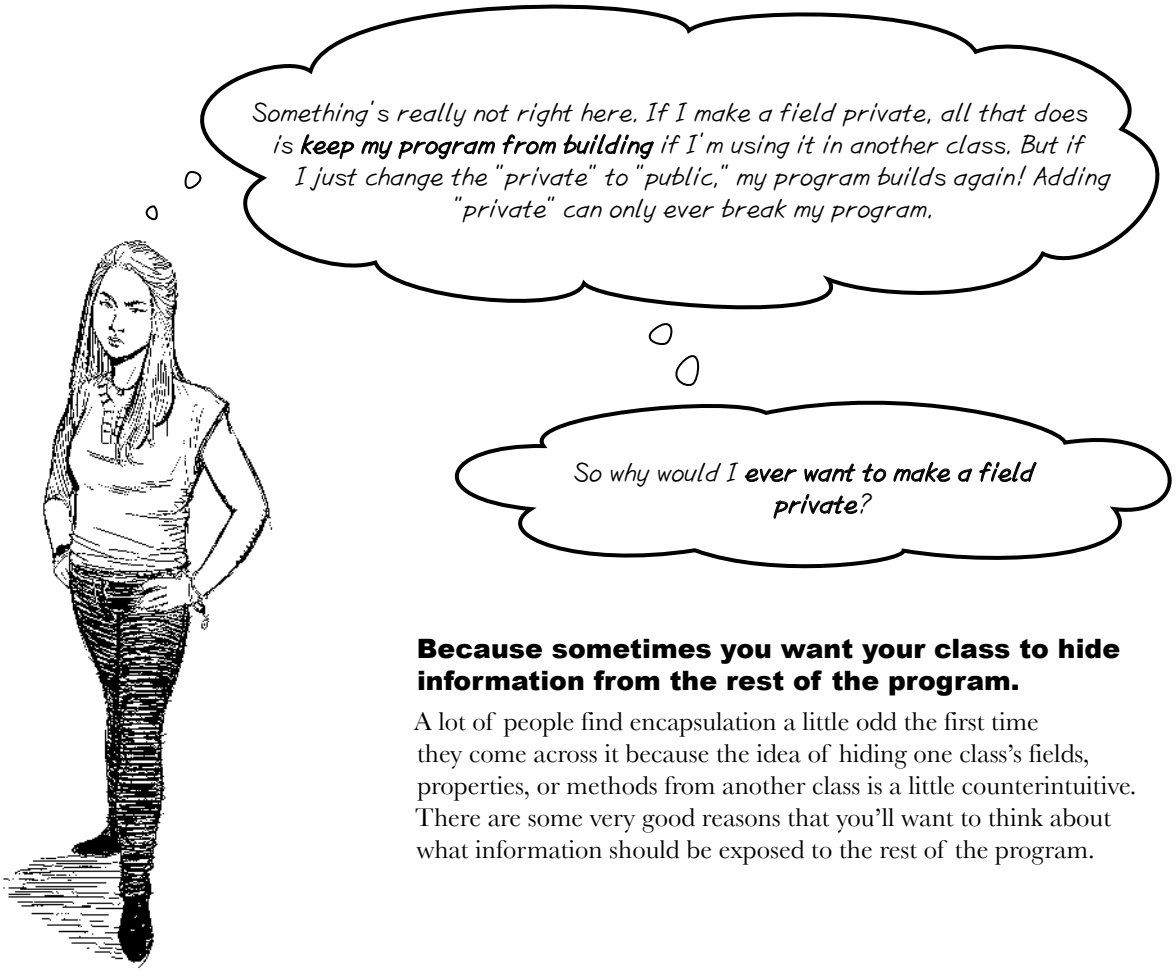
```
static class HiLoGame
{
    public const int MAXIMUM = 10;
    private static int currentNumber = Random.Shared.Next(1, MAXIMUM + 1);
    private static int nextNumber = Random.Shared.Next(1, MAXIMUM + 1);
    private static int pot = 10;

    public static int GetPot() { return pot; }

    public static void Guess(bool higher)
    {
       if ((higher && nextNumber >= currentNumber) ||
              (!higher && nextNumber <= currentNumber))
       {
            Console.WriteLine("You guessed right!");
            pot++;
       }
       else
       {
           Console.WriteLine("Bad luck, you guessed wrong.");
           pot--;
       }
       currentNumber = nextNumber;
       nextNumber = Random.Shared.Next(1, MAXIMUM + 1);
       Console.WriteLine($"The current number is {currentNumber}");
    }

    public static void Hint()
    {
        int half = MAXIMUM / 2;
        if (nextNumber >= half)
            Console.WriteLine($"The current number is {currentNumber}," +
                              $" the next number is at least {half}");
        else Console.WriteLine($"The current number is {currentNumber}," +
                               $" the next is at most {half}");
        pot--;
    }
}
```

The pot field is private, but the top-level statements can use the GetPot method to get its value *without having a way to modify it*.

The Hint method needs to be public because it's called from top-level statements. Notice how we didn't include the curly brackets for the `if/else` statement? An `if` or `else` clause that only has a single line *doesn't need brackets*.

You *protected* the pot field by making it private. It can only be modified by calling the Guess or Hint methods, and the GetPot method provides *read-only access*. This means the pot field is encapsulated inside the HiLoGame class: it can only be read—*but not written!*—from the top-level statements using the GetPot method.

This is an important point. Take a few minutes to really figure out how it works.



Because sometimes you want your class to hide information from the rest of the program.

A lot of people find encapsulation a little odd the first time they come across it because the idea of hiding one class's fields, properties, or methods from another class is a little counterintuitive. There are some very good reasons that you'll want to think about what information should be exposed to the rest of the program.

Encapsulation means having one class hide information from another. It helps you prevent bugs in your programs.

Watch it!

Encapsulation is not the same as security. Private fields are not secure.

*If you're building **a game with spies**, encapsulation is a great way to prevent bugs. If you're building **a program for real spies**, encapsulation is a terrible way to protect their data. For example, go back to your Hi-Lo game. Place a breakpoint on the first line of Program.cs, add a watch for* **Random.Shared***, and debug the program. If you **expand Static Members >> Non-Public Members >> LocalRandom >> Non-Public Members** you can see the internals of the class, including its seed values, or a initial values that it uses to generate its pseudorandom numbers. In fact, it's actually possible (but not straightforward!) to use these seed values to predict the next random number.*

It's not just the IDE that can see your objects' **`privates`**. *.NET has a tool called **reflection** that lets you write code to access objects in memory and look at their contents, even private fields. Here's a quick example of how it works. **Create a new console app** and add a class called HasASecret:*

```
class HasASecret
{
    // This class has a secret field. Does the private keyword make it secure?
    private string secret = "xyzzy";
}
```

*The reflection classes are in the **System.Reflection namespace**, so add this* **`using`** *statement to Program.cs, just above the top-level statements:*

```
using System.Reflection;
```

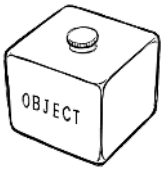
*Here are the top-level statements for Program.cs—they create a new instance of HasASecret, and then use reflection to read its **secret** field. It calls the GetType method, which is a method that you can call from any object to get information about its type:*

```
using System.Reflection;

HasASecret keeper = new HasASecret();

// Uncommenting this Console.WriteLine statement causes a compiler error:
// 'HasASecret.secret' is inaccessible due to its protection level
// Console.WriteLine(keeper.secret);

// But we can still use reflection to get the value of the secret field
FieldInfo[] fields = keeper.GetType().GetFields(
                BindingFlags.NonPublic | BindingFlags.Instance);

// This foreach loop will cause "xyzzy" to be printed to the console
foreach (FieldInfo field in fields)
{
    Console.WriteLine(field.GetValue(keeper));
}
```

Why encapsulation? Think of an object as an opaque box...

Sometimes you'll hear a developer refer to an object as an **opaque box**, and that's a pretty good way of thinking about objects in general. When we say something is an opaque box, we're saying that we can see how it behaves, but we have no way of knowing how it actually works.

When you call an object's method, you don't really care how that method works—at least, not right now. All you care about is that it takes the inputs you gave it and does the right thing.



When developers talk about an "opaque box," we mean something that hides any internal mechanisms so you don't need to know how it works to use it. If it just does one thing, and you don't need to give it any parameters, it's the code equivalent of a box with a single button on it.

You *could* include a lot more controls, like a window that shows you what's going on inside the box, and knobs and dials that let you muck with its internals. But if they don't actually do anything that your system needs, then they don't do you any good and can only cause problems.

Encapsulation makes your classes...

- **Easier to use**
  You already know that classes use fields to keep track of their state. Many of them use methods to keep those fields up-to-date—methods that no other class will ever call. It's pretty common to have a class that has fields, methods, and properties that will never be called by any other class. If you make those members private, then they won't show up in the IntelliSense window later when you need to use that class. Less clutter in the IDE will make your class easier to use.
- **Less prone to bugs**
  That bug in Owen's program happened because the app accessed a method directly rather than letting the other methods in the class call it. If that method had been private, we could have avoided that bug.
- **Flexible**
  A lot of times, you'll want to go back and add features to a program you wrote a while ago. If your classes are well-encapsulated, then you'll know exactly how to use them, and it will be easier to modify them later..

Brain Power

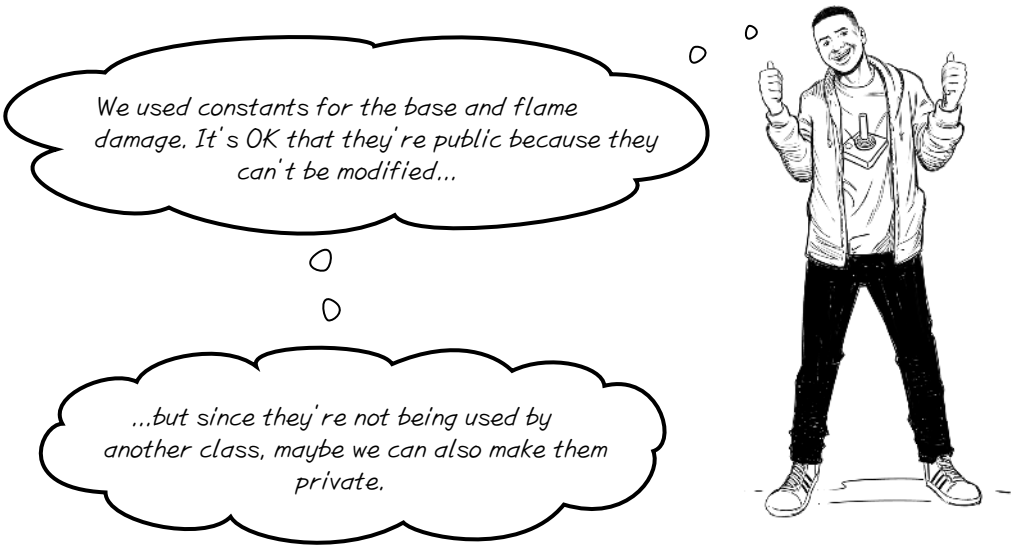
How could building a poorly encapsulated class now make your programs harder to modify later?

A few ideas for encapsulating classes

- **Is everything in your class public?**
  If your class has nothing but public fields and methods, you probably need to spend a little more time thinking about encapsulation.

- **Think about ways fields and methods can be misused.**
  What can go wrong if they're not set or called properly?

- **What fields require some processing or calculation to happen when they're set?**
  Those are prime candidates for encapsulation. If someone writes a method later that changes the value in any one of them, it could cause problems for the work your program is trying to do.



- **Only make fields and methods public if you need to.**
  If you don't have a reason to declare something public, don't—you could make things really messy for yourself by making all of the fields in your program public. But don't just go making everything private either. Spending a little time up front thinking about which fields really need to be public and which don't can save you a lot of time later.

A well-encapsulated class does **exactly the same thing** as one that has poor encapsulation!

Exactly! The difference is that the well-encapsulated one is built in a way that prevents bugs and is easier to use.

It's easy to take a well-encapsulated class and turn it into a poorly encapsulated class: do a search and replace to change every occurrence of `private` to `public`.

And that's a funny thing about the `private` keyword: you can generally take any program and do that search and replace, and it will still compile and work in exactly the same way. That's one reason that encapsulation can be a little difficult for some programmers to really "get" when they first see it.

When you come back to code that you haven't looked at in a long time, it's easy to forget how you intended it to be used. That's where encapsulation can make your life a lot easier!

This book so far has been about making programs **do things**—perform certain behaviors. Encapsulation is a little different. It doesn't change the way your program behaves. It's more about the "chess game" side of programming: by hiding certain information in your classes when you design and build them, you set up a strategy for how they'll interact later. The better the strategy, the **more flexible and maintainable** your programs will be, and the more bugs you'll avoid.

And just like chess, there are an almost unlimited number of possible encapsulation strategies!

If you encapsulate your classes well today, that makes them a lot easier to reuse tomorrow.

Bullet Points

- Always **think about what caused a bug** before you try to fix it. Take the time to really understand what's going on.
- Adding statements that print lines of output can be an effective debugging tool. Use **Debug.WriteLine** when you add statements to print diagnostic information.
- A **constructor** is a method that the CLR calls when it first creates a new instance of an object.
- **String interpolation** makes string concatenation more readable. Use it by adding a $ in front of a string and including values in {curly brackets}.
- The System.Console class writes its output to **standard streams** that provide the input and output for console apps.
- The System.Diagnostics.Debug class writes its output to **trace listeners**—special classes that perform specific actions with diagnostic output—including one that writes that output to the IDE's Output (Windows) or Application Output (macOS) window.
- People won't always use your classes in exactly the way you expect. **Encapsulation** is a technique for making your class members flexible and difficult to misuse.
- Encapsulation usually involves using the **`private`** keyword to keep some of the fields or methods in a class private so they can't be misused by other classes.
- When a class protects its data and provides members that are safe to use and difficult to misuse, we say that it's **well-encapsulated**.

*OK, we know the code for our sword damage app has some problems, what can we **do about it**?*

SwordDamage
Roll MagicMultiplier Damage
CalculateDamage SetMagic SetFlaming

Remember how you used Debug.WriteLine earlier in the chapter to sleuth out the bug in your app? You discovered that the SwordDamage class only works if its methods are called in a very specific order. This chapter is all about encapsulation, so it's a pretty safe bet that you'll use encapsulation later in the chapter to fix this problem. But...how, exactly?

Let's use encapsulation to improve the SwordDamage class

We just covered some great ideas for encapsulating classes. Let's see if we can start to apply those ideas to the SwordDamage class to keep it from being confused, misused, and abused by any app that we include it in.

Is every member of the SwordDamage class public?

Yes, indeed. The three fields (Roll, MagicMultiplier, and Damage) are public, and so are the three methods (CalculateDamage, SetMagic, and SetFlaming). We could stand to think about encapsulation.

Are fields or methods being misused?

Absolutely. In the first version of the damage calculator app, we called CalculateDamage when we should have just let the SetFlaming method call it. Even our attempt to fix it failed because we misused the methods by calling them in the wrong order.

Is there calculation required after setting a field?

Certainly. After setting the Roll field, we really want the instance to calculate damage immediately.

So what fields and methods really need to be public?

That's a great question. Take some time to think about the answer. We'll tackle it at the end of the chapter.

Making members of a class private can prevent bugs caused by other classes calling its public methods or updating its public fields in unexpected ways.

Brain Power

Think about those questions, then take another look at how the SwordDamage class works. What would you do to fix the SwordDamage class?

Encapsulation keeps your data safe

We've seen how the `private` keyword **protects class members from being accessed directly**, and how that can prevent bugs caused by other classes calling methods or updating fields in ways we didn't expect—like how your GetPot method in the Hi-Lo game gave read-only access to the private pot field, and only the Guess or Hint methods could modify it. This next class works in exactly the same way.

Let's use encapsulation in a class

Let's build a PaintballGun class for a paintball arena video game. The player can pick up magazines of paintballs and reload at any time, so we want the class to keep track of the total number of balls the player has *and* the number of balls currently loaded. We'll add a method to check if the gun is empty and needs to be reloaded. We also want it to keep track of the magazine size. Any time the player gets more ammo we want the gun to automatically reload a full magazine, so we'll make sure that always happens by providing a method to set the number of balls that calls the Reload method.

```
class PaintballGun
{
    public const int MAGAZINE_SIZE = 16;

    private int balls;
    private int ballsLoaded;

    public int GetBallsLoaded() { return ballsLoaded; }

    public bool IsEmpty() { return ballsLoaded == 0; }

    public int GetBalls() { return balls; }

    public void SetBalls(int numberOfBalls)
    {
        if (numberOfBalls > 0)
            balls = numberOfBalls;
        Reload();
    }

    public void Reload()
    {
        if (balls > MAGAZINE_SIZE)
            ballsLoaded = MAGAZINE_SIZE;
        else
            ballsLoaded = balls;
    }

    public bool Shoot()
    {
        if (ballsLoaded == 0) return false;
        ballsLoaded--;
        balls--;
        return true;
    }
}
```

We'll keep this constant public because it's going to be used by the top-level statements.

When the game needs to display the number of balls left and the number of balls loaded in the UI, it can call the GetBalls and GetBallsLoaded methods.

The game needs to be able to set the number of balls. The SetBalls method *protects the balls field* by only allowing the game to set a positive number of balls. Then it calls Reload to *automatically reload the gun*.

The only way to reload the gun is to call the Reload method, which loads the gun with a full magazine, or the remaining number of balls if there isn't a full magazine's worth. This keeps the balls and ballsLoaded fields from *getting out of sync*.

The Shoot method returns true and decrements the balls field if the gun is loaded, or false if it isn't.

Does the IsEmpty method make code that calls this class easier to read? Or is it redundant? There's no right or wrong answer—you could argue either side.

Write a console app to test the PaintballGun class

Do this!

Let's try out our new PaintballGun class. **Create a new console app called Paintball** and add the PaintballGun class to it. Here are the top-level statements—it uses a loop to call the various methods in the class:

```
PaintballGun gun = new PaintballGun();
while (true)
{
    Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
    if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
    Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
    char key = Console.ReadKey(true).KeyChar;
    if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
    else if (key == 'r') gun.Reload();
    else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
    else if (key == 'q') return;
}
```

This sort of console app should be pretty familiar by now. Take the time to read the code and understand how it works.

Click on PaintballGun and choose the "Generate class 'PaintballGun' in new file" quick action. The IDE will add a file called PaintballGun.cs to your project that contains an empty class called PaintballGun, but without a namespace declaration so it's in the global namespace.

Our class is well-encapsulated, but...

The class works, and we encapsulated it pretty well. The **balls field is protected**: it doesn't let you set a negative number of balls, and it stays in sync with the ballsLoaded field. The Reload and Shoot methods work as expected, and there don't seem to be any *obvious* ways we could accidentally misuse this class.

But have a closer look at this line from the top-level statements:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

Let's be honest—that's a downgrade from a field. If we still had a field, we could use the += operator to increase it by the magazine size. Encapsulation is great, but we don't want it to make our class annoying or difficult to use.

Is there a way to keep the balls field protected but still get the convenience of +=?

Use different cases for private and public fields

We used camelCase for the private fields and PascalCase for the public ones. PascalCase means capitalizing the first letter in every word in the variable name. camelCase is similar to PascalCase, except that the first letter is lowercase. It's called camelCase because it makes the uppercase letters look like "humps" of a camel.

Using different cases for public and private fields is a convention a lot of programmers follow. Your code is easier to read if you use consistent case when choosing names for fields, properties, variables, and methods.

Properties make encapsulation easier

So far you've learned about two kinds of class members, methods and fields. There's a third kind of class member that helps you encapsulate your classes: the **property**. A property is a class member that ***looks like a field*** when it's used, but it ***acts like a method*** when it runs.

A property is declared just like a field, with a type and a name, except instead of ending with a semicolon it's followed by curly brackets. Inside those brackets are **property accessors**, or methods that either return or set the property value. There are two types of accessors:

- A **get property accessor**, usually just referred to as a **get accessor** or **getter**, that returns the value of the property. It starts with the **`get`** keyword, followed by a method inside curly brackets. The method must return a value that matches the type in the property declaration.
- A **set property accessor**, usually just referred to as a **set accessor** or **setter**, that sets the value of the property. It starts with the **`set`** keyword, followed by a method inside curly brackets. Inside the method, the **`value`** keyword is a read-only variable that contains the value being set.

It is very common for a property to get or set a **backing field**, which is what we call a private field that's encapsulated by restricting access to it through a property.

Replace the GetBalls and SetBalls methods with a property

Replace this!

Here are the GetBalls and SetBalls methods from your PaintballGun class:

```
public int GetBalls() { return balls; }

public void SetBalls(int numberOfBalls)
{
    if (numberOfBalls > 0)
        balls = numberOfBalls;
    Reload();
}
```

This is *refactoring*. You're modifying the PaintballGun class so it behaves the same way but is structured differently.

Let's replace them with a property. **Delete both methods**.

Then **add this Balls property**:

```
public int Balls
{
    get { return balls; }

    set
    {
        if (value > 0)
            balls = value;
        Reload();
    }
}
```

This is the *declaration*. It says that the name of the property is Balls, and its type is int.

The *get accessor* (or *getter*) is identical to the GetBalls method that it replaced.

The *set accessor* (or *setter*) is almost identical to the SetBalls method. The only difference is it uses the `value` keyword where SetBalls used its parameter. The `value` keyword will always contain the value being assigned by the set accessor.

The old SetBalls method took an int parameter called numberOfBalls with the new value for the backing field. The setter uses the "value" keyword everywhere the SetBalls method used numberOfBalls.

Modify your top-level statements to use the Balls property

Now that you've replaced the GetBalls and SetBalls methods with a single property called Balls, your code won't build anymore. You need to update the top-level statements to use the Balls property instead of the old methods.

Update this!

The GetBalls method was called in this `Console.WriteLine` statement:

```
Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
```

You can fix that by **replacing `GetBalls()` with `Balls`**—when you do this, the statement will work just like before. Let's have a look at the other place where GetBalls and SetBalls were used:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

This was that messy line of code that looked ugly and clunky. Properties are really useful because they work like methods but you use them like fields. So let's use the Balls property like a field—**replace that line** with this statement that uses the += operator exactly like it would if Balls were a field:

```
else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
```

If Balls were a field, this is how you would use the += operator to update it. You use properties exactly the same way.

Here are the updated top-level statements:

```
PaintballGun gun = new PaintballGun();
while (true)
{
    Console.WriteLine($"{gun.Balls} balls, {gun.GetBallsLoaded()} loaded");
    if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
    Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
    char key = Console.ReadKey(true).KeyChar;
    if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
    else if (key == 'r') gun.Reload();
    else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
    else if (key == 'q') return;
}
```

Debug your PaintballGun class to understand how the property works

Use the debugger to really get a good sense of how your new Ball property works:

- Place a breakpoint inside the curly brackets of the get accessor (`return balls;`).
- Place another breakpoint on the first line of the set accessor (`if (value > 0)`).
- Place a breakpoint at the top of *Program.cs* and start debugging. Step over each statement.
- When you step over Console.WriteLine, the debugger will hit the breakpoint in the getter.
- Keep stepping over methods. When you execute the **+=** statement, the debugger will hit the breakpoint in the setter. Add a watch for the backing field **balls** and the **`value`** keyword.

Auto-implemented properties simplify your code

Add this!

A very common way to use a property is to create a backing field and provide get and set accessors for it. Let's create a new BallsLoaded property that **uses the existing ballsLoaded field** as a backing field:

```
private int ballsLoaded;

public int BallsLoaded {
   get { return ballsLoaded; }
   set { ballsLoaded = value; }
}
```

This property uses a private *backing field*. Its getter returns the value in the field, and its setter updates the field.

Now you can **delete the GetBallsLoaded method** and modify your top-level statements to use the property:

```
Console.WriteLine($"{gun.Balls} balls, {gun.BallsLoaded} loaded");
```

Run your program again. It should still work exactly the same way.

VSCode doesn't support snippets yet, so if you're using VSCode just type in the whole property.

Use the prop snippet to create an auto-implemented property

An **auto-implemented property**—sometimes called an **automatic property** or **auto-property**—is a property that has a getter that returns the value of the backing field, and a setter that updates it. In other words, it works just like the BallsLoaded property that you just created. There's one important difference: when you create an automatic property ***you don't define the backing field***. Instead, the C# compiler creates the backing field for you, and the only way to update it is to use the get and set accessors.

Visual Studio gives you a really useful tool for creating automatic properties: a **code snippet**, or a small, reusable block of code that the IDE inserts automatically. Let's use it to create a BallsLoaded auto-property.

1. **Remove the BallsLoaded property and backing field.**
   Delete the BallsLoaded property you added, because we're going to replace it with an auto-implemented property. Then delete the ballsLoaded backing field (`private int ballsLoaded;`) too, because any time you create an automatic property the C# compiler generates a hidden backing field for you.

2. **Tell the IDE to start the prop snippet.**
   Put your cursor where the field used to be, and then **type `prop` and press the Tab key twice** (or use the menu like you did in Chapter 2) to tell the IDE to start a snippet. It will add this line to your code:

   ```
   public int MyProperty { get; set; }
   ```

   VSCode may not support snippets, so just type in the whole property.

   The snippet is a template that lets you edit parts of it—the prop snippet lets you edit the type and the property name. Press the Tab key once to switch to the property name, then **change the name to `BallsLoaded`** and press Enter to finalize the snippet:

   ```
   public int BallsLoaded { get; set; }
   ```

   You don't have to declare a backing field for an automatic property because the C# compiler creates it automatically.

3. **Fix the rest of the class.**
   Since you removed the ballsLoaded field, your PaintballGun class doesn't compile anymore. That has a quick fix—the **b**allsLoaded field appears five times in the code (once in the IsEmpty method, and twice in the Reload and Shoot methods). Change them to **B**allsLoaded—now your app works again.

Use a private setter to create a read-only property

Let's take another look at the auto-implemented property that you just created:

```
public int BallsLoaded { get; set; }
```

This is definitely a great replacement for a property with get and set accessors that just update a backing field. It's more readable and has less code than the ballsLoaded field and GetBallsLoaded method. So that's an improvement, right?

But there's one problem: ***we've broken the encapsulation***. The whole point of the private field and public method was to make the number of balls loaded read-only. The top-level statements could easily set the BallsLoaded property. We made the field private and created a public method to get the value so that it could only be modified from inside the PaintballGun class.

Make the BallsLoaded setter private

Luckily, there's a quick way to make our PaintballGun class well-encapsulated again. When you use a property, you can put an access modifier in front of the `get` or `set` keyword.

You can make a **read-only property** that can't be set by another class by making its set accessor **private**. In fact, you can leave out the setter entirely and just have a getter, even on an automatic property—so your BallsLoaded property *could* look like this: `public int BallsLoaded { get; }`

In this app we want to encapsulate BallsLoaded so only the PaintballGun class can set its value but any other class can read it. So go ahead and **make the set accessor private**:

```
public int BallsLoaded { get; private set; }
```

You can make your automatic property read-only by making its setter private.

Now the BallsLoaded field is a **read-only property**. It can be read anywhere, but it can only be updated from inside the PaintballGun class. The PaintballGun class is well-encapsulated again.

We made BallsLoaded a read-only property with a private setter so that other classes ***can't accidentally set it****, right?*

Exactly. Making a set accessor private is a great way to improve encapsulation for a class.

Think about how a paintball gun would work in a paintball arena game. Usually, there are only two things that affect the ammo loaded in the gun: the player loading or firing it. If we design the app so that only the PaintballGun class can modify its BallsLoaded property, it means we won't accidentally update it from any other class in the system. That's a great way to prevent bugs.

What if we want to change the magazine size?

Right now the PaintballGun class uses a `const` for the magazine size:

Replace this!

```
public const int MAGAZINE_SIZE = 16;
```

What if we want the game to set the magazine size when it instantiates the gun? Let's **replace it with a property**.

1. **Remove the MAGAZINE_SIZE constant and replace it with a read-only property.**

```
public int MagazineSize { get; private set; }
```

2. **Modify the Reload method to use the new property.**

```
if (balls > MagazineSize)
    BallsLoaded = MagazineSize;
```

3. **Fix the line in the top-level statements that adds the ammo.**

```
else if (key == '+') gun.Balls += gun.MagazineSize;
```

But there's a problem...how do we initialize MagazineSize?

It's called a field initializer, even if you're using it to initialize a property.

The MAGAZINE_SIZE constant used to be set to 16. Now we've replaced it with an auto-property, and if we want, we can initialize it to 16 just like a field by adding a **field initializer** to the end of the declaration:

```
public int MagazineSize { get; private set; } = 16;
```

Now the MagazineSize property will be initialized to 16 as soon as the game starts. We can even change this line to use Random.Shared.Next to initialize the magazine with a random number of paintballs:

```
public int MagazineSize { get; private set; } = Random.Shared.Next(8, 17);
```

But let's say we wanted to do something a little different—like initialize it to the value of the BallsLoaded property:

```
public int MagazineSize { get; private set; } = BallsLoaded;
```

```
int PaintballGun.BallsLoaded { get; private set; }

CS0236: A field initializer cannot reference the non-static field, method, or property 'PaintballGun.BallsLoaded'
```

Now you get a compiler error. You can only use a literal, constant, or static field, method, or property in a field initializer. Random.Shared is a static property, so you're allowed to use it in your field initializer. But you can't use the BallsLoaded property, because **it's not static**.

Go ahead and **remove the field initializer**:

```
public int MagazineSize { get; private set; };
```

What if you want the game to decide how many balls to load in a new gun depending on the current level? How would you initialize the MagazineSize property if that data isn't available in a static field or property?

Use a constructor with parameters to initialize properties

You saw earlier in the chapter that you can initialize an object with a constructor, or a special method that's called when the object is first instantiated. Constructors are just like any other method—which means they can have **parameters**. We'll use a constructor with parameters to initialize the properties.

The constructor in your MAUI app's MainPage class looks like this: `public MainPage()` – that's a **parameterless constructor**, so just like any other method without parameters, the declaration ends with `()`. Now let's **add a constructor with parameters** to the PaintballGun class. Here's the constructor to add:

Add a constructor to a class by creating a method that has the same name as the class and no return type.

This constructor takes three *parameters*, an int called `balls`, an int called `magazineSize`, and a bool called `loaded`.

```
public PaintballGun(int balls, int magazineSize, bool loaded)
{
    this.balls = balls;
    MagazineSize = magazineSize;
    if (loaded) Reload();
}
```

The constructor runs as soon as a new instance is created, so we put code into the body of the method to set the number of balls and magazine size, and reload the gun if needed. Notice the `this` *keyword* in the first line. Why do you think we need to use it?

Uh-oh—there's a problem. As soon as you add the constructor, the IDE will tell you that there's an error:

CS7036	There is no argument given that corresponds to the required formal parameter 'balls' of 'PaintballGun.PaintballGun(int, int, bool)'

What do you think we need to do to fix this error?

When a parameter has the same name as a field, it *masks* the field.

The constructor's `balls` *parameter has the same name as the field called balls. Since they have the same name, the parameter takes precedence inside the body of the constructor. That's called* ***masking****—when a parameter or a variable in a method has the same name as a field, using that name in the method refers to the parameter or variable, not the field. That's why we need to use the* **`this`** *keyword in the PaintballGun constructor:*

```
this.balls = balls;
```

When we just use `balls`*, it refers to the parameter. We want to set the field, and since it has the same name, we need to use* `this.balls` *to refer to the field.*

And by the way, this doesn't just apply to constructors. It's true for ***any*** *method.*

Specify arguments when you use the new keyword

When you added the constructor, the IDE told you that the top-level statements have an error on the `new` statement (`PaintballGun gun = new PaintballGun()`). Here's what that error looks like:

```
CS7036  There is no argument given that corresponds to the required formal
        parameter 'balls' of 'PaintballGun.PaintballGun(int, int, bool)'
```

Read the text of the error—it's telling you exactly what's wrong. Your constructor now takes arguments, so it needs parameters. Start typing the `new` statement again, and the IDE will tell you exactly what you need to add:

```
PaintballGun gun = new PaintballGun()
                   PaintballGun(int balls, int magazineSize, bool loaded)
```

You've been using `new` to create instances of classes. So far, all of your classes have had parameterless constructors, so you never needed to provide any arguments.

Now you have a **constructor with parameters**, and like any method with parameters, it requires you to specify arguments with types that match those parameters.

Let's modify your top-level statements to **pass parameters to the PaintballGun constructor**.

Modify this!

1. **Add the ReadInt method that you wrote for Owen's ability score calculator in Chapter 4.**
   You need to get the arguments for the constructor from somewhere. Luckily, you already have a perfectly good method that prompts the user for int values, so it makes sense to reuse it here. Owen's ability score app had a static method called ReadInt that reads an int value from the console, taking a default value and prompt as parameters. Copy the method and **paste it into your top-level statements**.

2. **Add code to read values from the console input.**
   Now that you've added the ReadInt method from Chapter 4, you can use it to get two int values. Add these four lines of code to the top of your top-level statements:

```
int numberOfBalls = ReadInt(20, "Number of balls");
int magazineSize = ReadInt(16, "Magazine size");

Console.Write($"Loaded [false]: ");
bool.TryParse(Console.ReadLine(), out bool isLoaded);
```

If TryParse can't parse the line, it will leave isLoaded with the default value, which for a bool is false.

3. **Update the `new` statement to add arguments.**
   Now that you have values in variables with types that match the parameters in the constructor, you can update the `new` statement to pass them to the constructor as arguments:

```
PaintballGun gun = new PaintballGun(numberOfBalls, magazineSize, isLoaded);
```

4. **Run your program.**
   Now run your program. It will prompt you for the number of balls, the magazine size, and whether or not the gun is loaded. Then it will create a new instance of PaintballGun, passing arguments to its constructor that match your choices.

Sharpen your pencil

Here's the **final version of the PaintballGun class**. You did a lot of refactoring, and now it looks really different than the original version. Go back through all of the steps that you followed, compare the original code for the PaintballGun class with the final version, and write down everything that changed.

```
class PaintballGun
{
    private int balls;

    public PaintballGun(int balls, int magazineSize, bool loaded)
    {
        this.balls = balls;                          .....
        MagazineSize = magazineSize;                 .....
        if (loaded) Reload();                        .....
    }

    public int MagazineSize { get; private set; }    .....

    public int BallsLoaded { get; private set; }

    public bool IsEmpty() { return BallsLoaded == 0; }   .....

    public int Balls                                     .....
    {                                                    .....
        get { return balls; }

        set
        {                                            .....
            if (value > 0)                           .....
                balls = value;                       .....
            Reload();                                .....
        }
    }

    public void Reload()
    {
        if (balls > MagazineSize)
            BallsLoaded = MagazineSize;              .....
        else                                         .....
            BallsLoaded = balls;                     .....
    }

    public bool Shoot()
    {
        if (BallsLoaded == 0) return false;          .....
        BallsLoaded--;                               .....
        balls--;                                     .....
        return true;
    }
}
```

Sharpen your pencil Solution

Sens-AI

To get the solution to this "Sharpen your pencil" exercise, use AI to analyze the original and refactored versions of the classes and tell you what was refactored. Open ChatGPT, Copilot, Bard, or another AI chatbot and give it this prompt:

```
The PaintballGun class was refactored. Describe the changes.
Here's the code for the original version of the Paintball Class:
(paste the original code that we gave you here)
Here's the code for the refactored version of the Paintball Class:
(copy the refactored code from your IDE and paste it here)
```

Give this same prompt to multiple AI chatbots and compare their answers with yours. Did they all agree?

Here's the final version of the Paintball app's top-level statements. Try using AI to describe the differences between the original version that we gave you and the final version pasted from your IDE.

```
int numberOfBalls = ReadInt(20, "Number of balls");
int magazineSize = ReadInt(16, "Magazine size");

Console.Write($"Loaded [false]: ");
bool.TryParse(Console.ReadLine(), out bool isLoaded);

PaintballGun gun = new PaintballGun(numberOfBalls, magazineSize, isLoaded);

while (true)
{
    Console.WriteLine($"{gun.Balls} balls, {gun.BallsLoaded} loaded");
    if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
    Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
    char key = Console.ReadKey(true).KeyChar;
    if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
    else if (key == 'r') gun.Reload();
    else if (key == '+') gun.Balls += gun.MagazineSize;
    else if (key == 'q') return;
}

static int ReadInt(int defaultValue, string prompt)
{
    Console.Write(prompt + " [" + defaultValue + "]: ");
    string? line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    }
    else
    {
        Console.WriteLine("   using default value " + defaultValue);
        return defaultValue;
    }
}
```

The new version of the top-level statements uses the constructor.

The Balls, BallsLoaded, and MagazineSize properties help to encapsulate the class.

Here's the ReadInt method that you copied from Owen's damage calculator app in Chapter 4.

Here's a quick coding challenge for you!

IsEmpty is still a method. Can you figure out how to turn it into a property?

there are no Dumb Questions

Q: We replaced methods with properties earlier. Is there a difference between how a method works and how a getter or setter works?

A: No. Get and set accessors are a special kind of method—they look just like a field to other objects, and are called whenever that "field" is set. Getters always return a value that's the same type as the field. A setter works just like a method with one parameter called `value` whose type is the same as the field.

Q: So you can have ANY kind of statement in a property?

A: Absolutely. Anything you can do in a method, you can do in a property—you can even include complicated logic that does anything you can do in a normal method. A property can call other methods, access other fields, and even create instances of objects. Just remember that they only get called when a property gets accessed, so they should only include statements that have to do with getting or setting the property.

Q: Why would I need complicated logic in a get or set accessor? Isn't it just a way of modifying fields?

A: Sometimes you know that every time you set a field, you'll have to do some calculation or perform some action. Think about Owen's problem—he ran into trouble because the app didn't call the SwordDamage methods in the right order after setting the Roll field. If we replaced all of the methods with properties, then we could make sure the setters do the damage calculation correctly. (In fact, you're about to do exactly that at the end of the chapter!)

Q: I'm pretty sure I get constructors, but can you explain them one more time just so I'm sure?

A: A **constructor** is a special method that's called ***automatically*** when a new instance of a class is created with the `new` keyword. If the constructor takes parameters, then it can pass arguments when you create the new instance. You just modified the PaintballGun constructor to take three parameters —`int balls`, `int magazineSize`, `bool loaded`—and when you did that you had to modify the code that instantiated the class to add three arguments between the parentheses after the class name.

Q: How do I know which method in a class is its constructor?

A: A constructor is declared as a method with ***no return type***, and the method name ***matches the class name***. If you see a method in a class that has no return type and its function name matches the class name, it's the constructor.

Q: Can a class have more than one constructor?

A: Yes. A class can have multiple constructors, as long as they take different parameters. Your app can figure out which constructor to call based on the arguments used in the `new` statement that instantiates the class.

Exercise

Use the debugger to really understand how the constructor works. Create a new console app and add this class with a constructor and a public field called i. (There's no answer to this exercise; it's just about exploring constructors.)

```
public class ConstructorTest
{
    public int i = Random.Shared.Next();

    public ConstructorTest(int newIntValue)
    {
        i = newIntValue;
    }
}
```

Add breakpoints to the statement in the constructor and on the closing bracket.

When the app hits the first breakpoint, i is initialized with its default value, 0. Continue (F5) to the next breakpoint, and i will have a random number. Continue again, and it will be updated with the value 2 passed to the constructor.

Add one breakpoint on the field declaration and two inside the constructor. Replace the top-level statements with a single statement to instantiate the class: **`new ConstructorTest(2);`**—then run your app and watch the value of i.

Use AI to help you learn about refactoring and encapsulation

One reason C# is such a flexible and useful programming language is that there are many different ways to do the same thing. And since there are so many ways to write a specific piece of code, you can choose the one that makes it easiest to understand. That's why **refactoring**—or modifying the *structure* of code without changing its *behavior*—is such a powerful tool. AI chatbots can help you refactor your code.

Ask an AI chatbot to refactor your code

Here's the AbilityScoreCalculator class that you worked on in Chapter 4:

```
internal class AbilityScoreCalculator
{
   public int RollResult = 14;
   public double DivideBy = 1.75;
   public int AddAmount = 2;
   public int Minimum = 3;
   public int Score;

   public void CalculateAbilityScore()
   {
      // Divide the roll result by the DivideBy field
      double divided = RollResult / DivideBy;

      // Add AddAmount to the result and round down
      int added = AddAmount + (int)divided;

      // If the result is too small, use Minimum
      if (added < Minimum)
          Score = Minimum;
      else
          Score = added;
   }
}
```

Read all code written by an AI before you use it.

AI can be a really powerful tool for generating code. It can be especially valuable for learning, so take the time to read and understand the code that the AI chatbot generated. We also learned an important lesson in Chapter 1: AIs sometimes make mistakes. ***Read*** **every line** ***of code*** *generated by an AI before you use it in a project. If you don't read all of the code, your app might do something you don't want it to do.*

Read through it again, but now think about what you've learned about properties, constructors, and encapsulation. Do you see opportunities to refactor it? There are definitely ways to do that.

Go to your favorite AI chatbot and give it this prompt:

> *Refactor the following C# class to encapsulate Score by using a property instead of a public field.*
>
> *(paste the entire AbilityScoreCalculator class into the chat)*

Compare the code that the chatbot generated carefully against the code that you gave it. What did it change? Did it change Score into a property? Is it protected from being accidentally modified by another class?

> *Make Minimum property. Set its value using a constructor, and make sure that it can't be modified after the class is created.*

Now compare the new code that the chatbot gave you. Did it do what you expected it to do? Open your project from Chapter 4 and replace AbilityScoreCalculator with this new code. Does it still work the same way?

Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make classes that will compile and run and produce output that matches the sample.

This program is a math quiz game that asks a series of random multiplication or addition questions and checks the answer. Here's what it looks like when you play it:

```
8 + 5 = 13
Right!
4 * 6 = 24
Right!
4 * 9 = 37
Wrong! Try again.
4 * 9 = 36
Right!
9 * 8 = 72
Right!
6 + 5 = 12
Wrong! Try again.
6 + 5 = 9
Wrong! Try again.
6 + 5 = 11
Right!
8 * 4 = 32
Right!
8 + 6 = Bye
Thanks for playing!
```

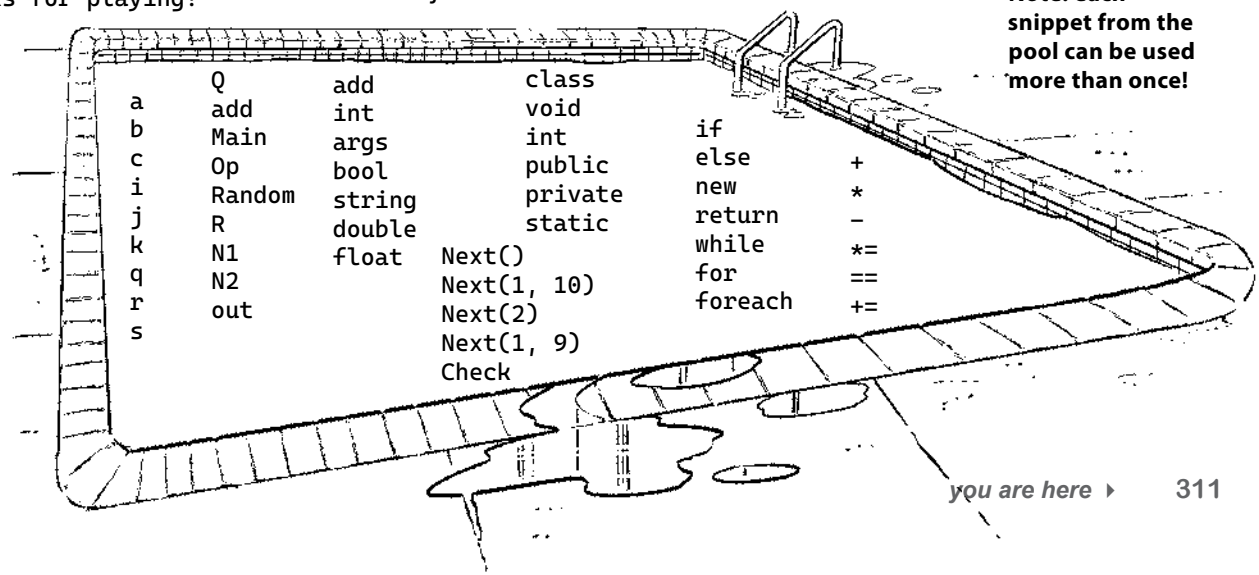
The game generates random addition or multiplication questions.

If you get a question wrong, it keeps asking until you get it right.

The game ends when you enter an answer that isn't a number.

```
Q _____ = _____ Q(_____.R._________ == 1);
while (true) {
    Console.Write($"{q._____} {q._____} {q._____} = ");
    if (!int.TryParse(Console.ReadLine(), out int i)) {
        Console.WriteLine("Thanks for playing!");
        __________;
    }
    if (_____._________(_____)) {
        Console.WriteLine("Right!");
        _____ = _____ Q(_____.R._________ == 1);
    }
    else {
        Console.WriteLine("Wrong! Try again.");
    }
}

class Q {
    public Q(bool add) {
        if (add) _____ = "+";
        else _____ = "*";
        N1 = _____._______________;
        N2 = _____._______________;
    }

    public __________ Random R = new Random();
    public __________ N1 { get; ___________ set; }
    public __________ Op { get; ___________ set; }
    public __________ N2 { get; ___________ set; }

    public __________ Check(int ______)
    {
        if (_____ == "+") return (a _____ N1 + N2);
        else return (a _____ _____ *  _____);
    }
}
```

Note: each snippet from the pool can be used more than once!



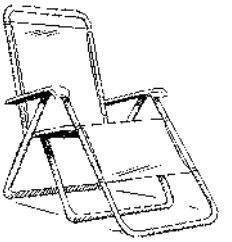
this puzzle is tough but you can do it!

```
Q __q__ = __new__ Q(__Q__.R.__Next(2)__ == 1);
while (true) {
  Console.Write($"{q.__N1__} {q.__Op__} {q.__N2__} =
");
  if (!int.TryParse(Console.ReadLine(), out int i)) {
     Console.WriteLine("Thanks for playing!");
     __return__;
  }
  if (__q__.__Check__(__i__)) {
     Console.WriteLine("Right!");
     __q__ = new Q(__Q__.R.__Next(2)__ == 1);
  }
      else Console.WriteLine("Wrong! Try again.");
}


class Q {
    public Q(bool add) {
        if (add) __Op__ = "+";
        else __Op__ = "*";
        N1 = __R__.__Next(1, 10)__;
        N2 = __R__.__Next(1, 10)__;
    }

    public __static__ Random R = new Random();
    public __int__ N1 { get; __private__ set; }
    public __string__ Op { get; __private__ set; }
    public __int__ N2 { get; __private__ set; }

    public __bool__ Check(int __a__)
    {
        if (__Op__ == "+") return (a __==__ N1 + N2);
        else return (a __==__ __N1__ * __N2__);
    }
}
```

Note: each snippet from the pool can be used more than once!

We put a check next to each snippet that was used in the solution. ✓

You can include classes in your *Program.cs* file along with the top-level statements. Try entering this entire app into your *Program.cs* file, including the Q class definition.



Pool Puzzle Solution

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make classes that will compile and run and produce output that matches the sample.

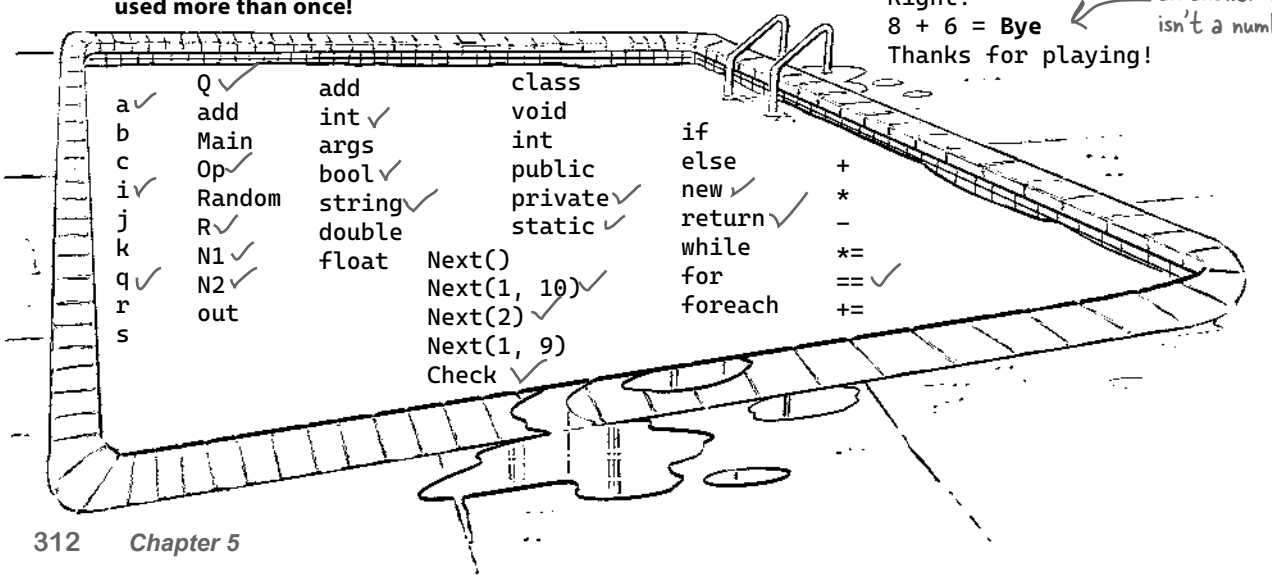
This program is a math quiz game that asks a series of random multiplication or addition questions and checks the answer. Here's what it looks like when you play it:

```
8 + 5 = 13
Right!
4 * 6 = 24
Right!
4 * 9 = 37
Wrong! Try again.
4 * 9 = 36
Right!
9 * 8 = 72
Right!
6 + 5 = 12
Wrong! Try again.
6 + 5 = 9
Wrong! Try again.
6 + 5 = 11
Right!
8 * 4 = 32
Right!
8 + 6 = Bye
Thanks for playing!
```

The game generates random addition or multiplication questions.

If you get a question wrong, it keeps asking until you get it right.

The game ends when you enter an answer that isn't a number.



Initialize fields and properties inline or in the constructor

We just saw in the Pool Puzzle that **you can include classes in your *Program.cs***—they just need to be declared below the top-level statements. Let's take advantage of that to explore a little more about how the constructor works. Create a new Console Application project and add this code to *Program.cs*:

```
new MessageWriter().WriteMessage();

class MessageWriter
{
    public string Message { get; set; }

    public void WriteMessage()
    {
        Console.WriteLine($"The message is: {Message}");
    }

}
```

The compiler will give you the warning about null values that we learned about in Chapter 4.

> ⚠ CS8618 Non-nullable property 'Message' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.

Take another look at that warning—it's telling you that the Message property must be set to a non-null value when it exits the constructor. Now that you've used constructors, that should make more sense.

Try giving the Message property a default value by adding an **inline assignment**:

```
public string Message { get; set; } = "This is a message";
```

The warning should disappear. That's because that assignment happens before the constructor is executed. Try experimenting with this by adding a constructor, then putting breakpoints in it and on the property assignment.

```
3   class MessageWriter
4   {
        1 reference
5       public string Message { get; set; } = "This is a message";
6
        1 reference
7       public MessageWriter()
8       {
9           Console.WriteLine("Running the constructor");
10      }
11
        1 reference
12      public void WriteMessage()
13      {
14          Console.WriteLine($"The message is: {Message}");
15      }
16
17  }
```

You can add a breakpoint on the property assignment by putting your cursor at the end of the line and pressing F9 to toggle the breakpoint.

VSCode might not highlight the inline assignment value in red like this Visual Studio screenshot, but it will still break on the assignment.

When you debug this app, the breakpoint on the property assignment gets hit first. When you continue, the breakpoint in the constructor gets hit next—so if you set a property in the constructor, it will overwrite any value you assigned inline.

This is a really good time to talk about aesthetics in video games. If you think about it, encapsulation doesn't really give you a way to do anything that you couldn't before. You could still write the same programs without properties, constructors, and private methods—but they would sure look really different. That's because not everything in programming is about making your code do something different. Often, it's about making your code do the same thing but in a better way. Think about that when you read about aesthetics. They don't change the way your game behaves, they change the way the player thinks and feels about the game.

Game Design...and Beyond

Aesthetics

How did you feel the last time you played a game? Was it fun? Did you feel a thrill, a rush of adrenaline? Did it give you a sense of discovery or accomplishment? Did you get a feeling of competition or cooperation with other players? Was there an engaging story? Was it funny? Sad? Games bring out emotional responses in us, and that's the idea behind aesthetics.

Does it seem weird to talk about feelings and video games? It shouldn't—emotions and feelings have always played an important part in game design, and the most successful games have an important aesthetic aspect to them. Think about that satisfying feeling you get when you drop a long piece in Tetris and it clears four rows of blocks. Or that rush in Pac-Man when Blinky (the red ghost) is just pixels behind you when you swallow the power pellet.

- It's obvious how **art and visuals, music and sound**, or story writing can influence aesthetics, but aesthetics is more than the artistic elements of a game. Aesthetics can come from the way the game is **structured**.
- You can find **aesthetics in tabletop games**. Poker is known for its emotional highs and lows, the feeling of pulling off a great bluff. Even a simple card game like Go Fish! has its own aesthetics: the growing back and forth, as players figure out each other's hands; the crescendo toward a winner, as players put each new book of cards on the table; the thrill of drawing that card you need; just saying "Go fish!" when asked for the wrong card.
- When a game provides a **challenge** it gives players obstacles to get past, creating a feeling of accomplishment and personal victory.
- The pure **tactile sensation** of a game—the beat of a rhythm game, the satisfying "gulp" of eating a power pellet, the "vroom" and blur of an accelerating car—provides pleasure.
- Playing a cooperative or multiplayer game brings a sense of **fellowship** with others.
- A game's **narrative** draws the player into the drama of a story. A game that provides **fantasy** not only transports a player to another world, but lets that player be another person (or nonperson!) entirely.
- Games with **expression** give the player self-discovery, a way to learn more about themselves.

Believe it or not, we can use these ideas behind aesthetics to learn a **larger lesson about development** that applies to any kind of program or app, not just a game. Let these ideas sink in for now—we'll return to this in the next chapter.

Some developers are really skeptical when they read about aesthetics because they assume that only the mechanics of the game matter. Here's a quick thought experiment to show how important aesthetics can be. Say you have two games with identical mechanics. There's just one very tiny difference between them. In one game, you're kicking boulders out of the way to save a village. In the other game, you're kicking puppies and kittens because you are a horrible person. Even if every other aspect of those games is identical, those are two very different games. That's the power of aesthetics.

Sharpen your pencil

This code has problems. It's supposed to be code for a simple gumball vending machine: you put in a coin and it dispenses gum. We've identified four specific problems that will cause bugs. Use the lines provided to write down what you think is wrong with the lines the arrows are pointing to.

```
class GumballMachine {
    private int gumballs;

                private int price;
                public int Price
                {
                    get
                    {
                      return price;
                    }
                }

        public GumballMachine(int gumballs, int price)
        {
                gumballs = this.gumballs;
                price = Price;
        }

    public string DispenseOneGumball(
            int price, int coinsInserted)
        {
                // check the price backing field
                if (this.coinsInserted >= price) {
                        gumballs -= 1;
                      return "Here's your gumball";
                } else {
                      return "Insert more coins";
                }
            }
    }
```

Sharpen your pencil Solution

This code has problems. We pointed out four specific lines that will cause bugs. Here's what's wrong with them.

Lowercase-p price refers to the constructor parameter, not the field. This line sets the PARAMETER to the value returned by the Price getter, but Price hasn't even been set yet, so it doesn't do anything useful. If you flip this around to set Price = price, it will work.

The "this" keyword is on the wrong "gumballs." this.gumballs refers to the property, while gumballs refers to the parameter.

This parameter masks the private field called price, and the comment says the method is supposed to be checking the value of the price backing field.

```
public GumballMachine(int gumballs, int price)
{
        gumballs = this.gumballs;
        price = Price;
}

public string DispenseOneGumball(int price, int coinsInserted)
{
        // check the price backing field
        if (this.coinsInserted >= price) {
                gumballs -= 1;
                return "Here's your gumball";
        } else {
                return "Insert more coins";
        }
}
```

The "this" keyword is on a parameter, where it doesn't belong. It should be on price, because that field is masked by a parameter.

It's worth your time to take an extra few minutes to ***really look at this code***. These are common mistakes that new programmers make when working with objects. If you learn to avoid them, you'll find it much more satisfying to write code.

there are no Dumb Questions

Q: If my constructor is a method, why doesn't it have a return type?

A: Your constructor doesn't have a return type because every constructor is always void—which makes sense, because there's no way for it to return a value. It would be redundant to make you type **void** at the beginning of each constructor.

Q: Can I have a getter without a setter?

A: Yes! When you have a get accessor but no set, you create a read-only property. For example, the SecretAgent class might have a public read-only field with a backing field for the name:

```
string spyNumber = "007";
public string SpyNumber {
   get { return spyNumber; }
}
```

Q: And I bet I can have a setter without a getter, right?

A: That's right! If you create a property with a setter but no getter, then your property **can only be written**. The SecretAgent class could use that for a property that other spies could write to, but not see:

```
public string DeadDrop {
  set {
    StoreSecret(value);
  }
}
```

Both of those techniques—set without get, or vice versa—can come in really handy when you're doing encapsulation.

One more thing—if you have an auto-property without a getter, it will return the default value for the type unless you give it a default value:

```
public int TwentyThree { get; } = 23;
```

Exercise

Use what you've learned about encapsulation to fix Owen's sword damage calculator. First, modify the SwordDamage class to replace the fields with properties and add a constructor. Once that's done, update the console app to use it. Finally, fix the MAUI app. (This exercise will go more easily if you create a new console app for the first two parts and a new MAUI app for the third.)

Part 1: Modify SwordDamage so it's a well-encapsulated class

1. Delete the Roll field and replace it with a property named Roll and a backing field named roll. The getter returns the value of the backing field. The setter updates the backing field, and then calls the CalculateDamage method.
2. Delete the SetFlaming method and replace it with a property named Flaming and a backing field named flaming. It works like the Roll property—the getter returns the backing field, the setter updates it and calls CalculateDamage.
3. Delete the SetMagic method and replace it with a property named Magic and a backing field named magic that works exactly like the Flaming and Roll properties.
4. Create an auto-implemented property named Damage with a public get accessor and private set accessor.
5. Delete the MagicMultiplier field. Modify the CalculateDamage method so it checks the property values for the Roll, Magic, and Flaming properties and does the entire calculation inside the method.
6. Add a constructor that takes the initial roll as its parameter. Now that the CalculateDamage method is only called from the property set accessors and constructor, there's no need for another class to call it. Make it private.
7. Add XML code documentation to all of the public class members.

Part 2: Modify the console app to use the well-encapsulated SwordDamage class

1. Add a method to *Program.cs* called RollDice that returns the results of a 3d6 roll.
2. Use the new RollDice method for the SwordDamage constructor argument and to set the Roll property.
3. Change the code that calls SetMagic and SetFlaming to set the Magic and Flaming properties instead.

Part 3: Modify the MAUI app to use the well-encapsulated SwordDamage class

1. Copy the code from Part 1 into a new MAUI app. Copy the XAML from the project earlier in the chapter.
2. Create a RollDice method that sets swordDamage.Roll to the results of a 3d6 roll, then calls DisplayDamage.
3. Make the Button_Clicked event handler call the RollDice method (and do nothing else).
4. Your SwordDamage class now has a constructor that takes one argument. Change the declaration so it looks like this (it doesn't matter what number you use, because the app will do a new random roll in the constructor):

```
SwordDamage swordDamage = new SwordDamage(3);
```

5. In the MainPage constructor, set the SwordDamage object's Flaming and Magic properties so they match the checkboxes. Then call the RollDice method to roll and display the damage.
6. Change the CheckedChanged event handlers for both checkboxes to use the Magic and Flaming properties instead of the old SetMagic and SetFlaming methods, then call DisplayDamage.
7. **Make your app more accessible.** Add SemanticProperties.Description properties to the checkbox labels and SemanticProperties.Hint properties to the checkboxes and buttons. Use the screen reader to test it.

Test everything. Use the debugger or Debug.WriteLine statements to make sure that it all REALLY works.

Exercise Solution

Now Owen finally has a class for calculating damage that's much easier to use without running into bugs. Each property recalculates the damage, so it doesn't matter what order you call them in. Here's the code for the well-encapsulated SwordDamage class:

```
class SwordDamage
{
    private const int BASE_DAMAGE = 3;
    private const int FLAME_DAMAGE = 2;
    private bool magic;
    private bool flaming;
    private int roll;

    /// <summary>
    /// Contains the calculated damage.
    /// </summary>
    public int Damage { get; private set; }

    /// <summary>
    /// Sets or gets the 3d6 roll.
    /// </summary>
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    /// <summary>
    /// True if the sword is magic, false otherwise.
    /// </summary>
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    /// <summary>
    /// True if the sword is flaming, false otherwise.
    /// </summary>
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }
```

Since these constants aren't going to be used by any other class, it makes sense to keep them private.

Here are the backing fields for the Roll, Magic, and Flaming.

The Damage property's private set accessor makes it read-only, so it can't be overwritten by another class.

Here's the Roll property with its private backing field. The set accessor calls the CalculateDamage method, which keeps the Damage property updated automatically.

The Magic and Flaming properties work just like the Roll property. They all call CalculateDamage, so setting any of them automatically updates the Damage property.

```
    /// <summary>
    /// Calculates the damage based on the current properties.
    /// </summary>
    private void CalculateDamage()
    {
        decimal magicMultiplier = 1M;
        if (Magic) magicMultiplier = 1.75M;

        Damage = BASE_DAMAGE;
        Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
        if (Flaming) Damage += FLAME_DAMAGE;
    }

    /// <summary>
    /// The constructor calculates damage based on default Magic
    /// and Flaming values and a starting 3d6 roll.
    /// </summary>
    /// <param name="startingRoll">Starting 3d6 roll</param>
    public SwordDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

All of the calculation is encapsulated inside the CalculateDamage method. It only depends on the get accessors for the Roll, Magic, and Flaming properties.

The constructor sets the backing field for the Roll property, then calls CalculateDamage to make sure the Damage property is correct.

Here's the code for the top-level statements in *Program.cs* for the console app:

```
SwordDamage swordDamage = new SwordDamage(RollDice());
while (true)
{
    Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                  "3 for both, anything else to quit: ");
    char key = Console.ReadKey().KeyChar;
    if (key != '0' && key != '1' && key != '2' && key != '3') return;
    swordDamage.Roll = RollDice();
    swordDamage.Magic = (key == '1' || key == '3');
    swordDamage.Flaming = (key == '2' || key == '3');
    Console.WriteLine($"\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\n");
  }

int RollDice()
{
    return Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7);
}
```

It made sense to move the 3d6 roll into its own method since it's called from two different places in the top-level statements. If you used "Generate method" to create it, the IDE made it private automatically.

Exercise Solution

Here's the code-behind in the *MainPage.xaml.cs* file:

It doesn't matter what we pass to the SwordDamage constructor, because we do a new roll in the page constructor. Is this the best way to handle this?

```
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage(3);

    public MainPage()
    {
        InitializeComponent();
        swordDamage.Flaming = Flaming.IsChecked;
        swordDamage.Magic = Magic.IsChecked;
        RollDice();
    }

    private void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
            + Random.Shared.Next(1, 7);
        DisplayDamage();
    }

    private void DisplayDamage()
    {
        Damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Flaming_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.Flaming = e.Value;
        DisplayDamage();
    }

    private void Magic_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.Magic = e.Value;
        DisplayDamage();
    }

    private void Button_Clicked(object sender, EventArgs e)
    {
        RollDice();
    }
}
```

The constructor sets the SwordDamage object's Flaming and Magic properties to match the checkboxes, then does an initial roll.

Brain Power

We had you pass 3 to the SwordDamage constructor because the MainPage constructor will call RollDice and set the Roll property to a new value. *This is a little messy!* Can you think of a better, more readable way to handle this?

Make the screen reader announce each roll

Do this!

Turn on the app, start the screen reader, and close your eyes. With the semantic properties, it should be pretty easy to use. Announcements can make a big difference for people who depend on a screen reader. Wouldn't it be great if it always announced the result every time you clicked the button to roll for damage?

Luckily, it's easy to do that! The **SemanticScreenReader.Default.Announce** method tells the screen reader to announce text. Add it to your Button_Clicked event handler method to announce the roll after the user clicks, then use your operating system's screen reader to test it out.

```
private void Button_Clicked(object sender, EventArgs e)
{
    RollDice();
    SemanticScreenReader.Default.Announce(Damage.Text);
}
```

Accessibility is important! You don't need to wait for us to tell you to add semantic properties or test your app with a screen reader. Thinking about accessibility early is a great way to improve your overall coding skills!

Exercise Solution

Here are the semantic properties we asked you to add to make the app more accessible:

```
<Label Text="Flaming"
        FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

<CheckBox x:Name="Flaming" Grid.Column="1"
            VerticalOptions="Center" HorizontalOptions="Start"
            SemanticProperties.Description="Determines if the sword is flaming"
            CheckedChanged="Flaming_CheckedChanged" Margin="20,0,0,0" />

<Label Text="Magic" Grid.Column="2"
        FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

<CheckBox x:Name="Magic" Grid.Column="3"
            VerticalOptions="Center" HorizontalOptions="Start"
            SemanticProperties.Description="Determines if the sword is magic"
            CheckedChanged="Magic_CheckedChanged" Margin="20,0,0,0" />

<Button Grid.Row="1" Grid.ColumnSpan="4" Margin="0,20,0,0"
        BorderColor="Black" BorderWidth="2"
        SemanticProperties.Hint="Throws a 3d6 roll for damage"
        Text="Roll for damage" FontSize="Medium" Clicked="Button_Clicked"/>

<Label x:Name="Damage" Grid.Row="2" Grid.ColumnSpan="4" Margin="0,20,0,0"
        BackgroundColor="Black" TextColor="White" FontSize="Large"
        SemanticProperties.Hint="The results of the damage roll"
        VerticalTextAlignment="Center" HorizontalTextAlignment="Center" />
```

A few useful facts about methods and properties

- **Every method in your class has a unique signature.**
  The first line of a method, which contains the access modifier, return value, name, and parameters is called the method's **signature**. Properties have signatures too—they consist of the access modifier, type, and name.

- **You can initialize properties in an object initializer.**
  You used object initializers before:

  ```
  Guy joe = new Guy() { Cash = 50, Name = "Joe" };
  ```

  You can also specify properties in an object initializer. If you do, the constructor is run first, then the properties are set. And you can only initialize public fields and properties in the object initializer.

- **Every class has a constructor, even if you didn't add one yourself.**
  The CLR needs a constructor to instantiate an object—it's part of the behind-the-scenes mechanics of how .NET works. So if you don't add a constructor to your class, the C# compiler automatically adds a parameterless constructor for you.

- **You can keep a class from being instantiated by other classes by adding a private constructor.**
  Sometimes you need to have really careful control over how your objects are created. One way to do that is to make your constructor private—then it can only be called from inside the class. Take a minute and try it out:

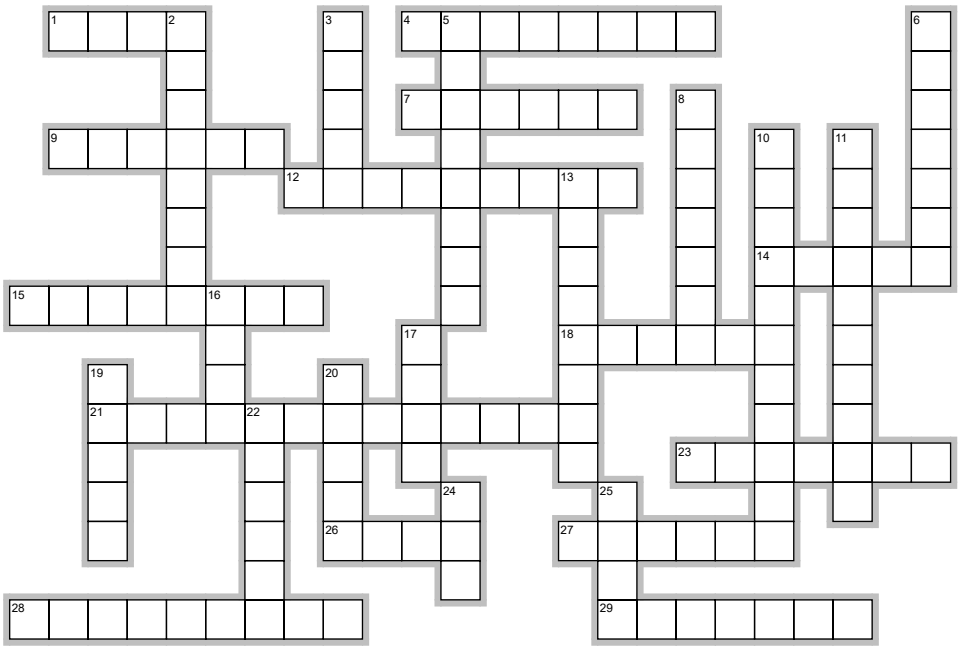
  ```
  class NoNew {
     private NoNew() { Console.WriteLine("I'm alive!"); }
     public static NoNew CreateInstance() { return new NoNew(); }
  }
  ```

  Add that NoNew class to a console app. If you try to add `new NoNew();` to your top-level statements, the C# compiler gives you an error ('*NoNew.NoNew()' is inaccessible due to its protection level*), but the **NoNew.CreateInstance** method creates a new instance just fine.

Objectcross

Take a break, sit back, and give your brain something to do. It's your standard crossword; all of the solution words are from the first five chapters of the book.

Crossword puzzle



Across

1. What (int) is doing in this line of code: x = (int) y;
4. Looks like a field but acts like a method
7. What kind of sequence is \n or \r?
9. If you want to create instances of a class, don't put this keyword in the declaration
12. A variable that points to an object
14. What an object is an instance of
15. The four whole-number types that only hold positive numbers
18. You can assign any value to a variable of this type
21. What you're doing when you use $ and curly braces to include values in a string
23. Draw one of these for your class before you start writing code
26. How you start a variable declaration
27. The numeric type that holds the biggest numbers
28. What you use to pass information into a method
29. If you want to store a currency value, use this type

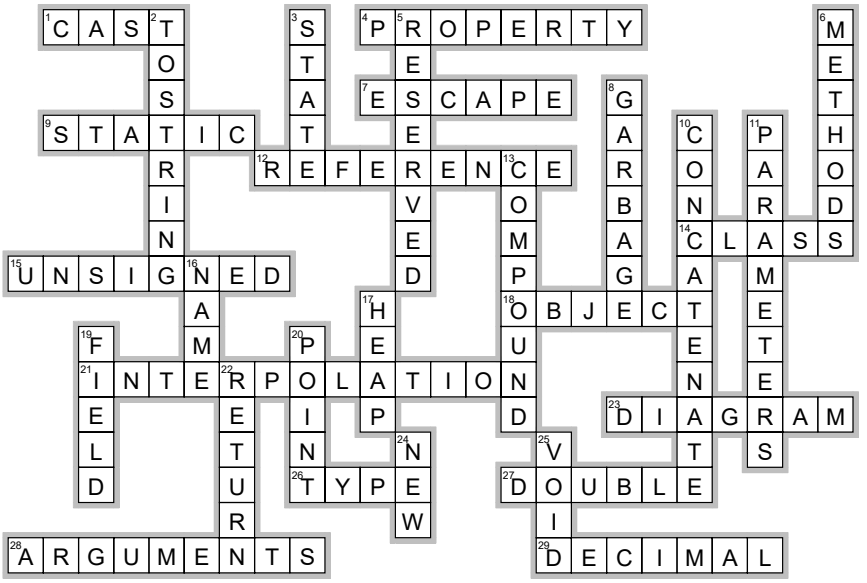
Down

2. Every object has this method that converts it to a string
3. An object's fields keep track of its _____
5. namespace, for, while, using, and new are examples of _____ keywords
6. These define the behavior of a class
8. The kind of collection that happens when the last reference to an object goes away
10. To _____ two strings, use the + operator
11. How a method tells you what to pass to it
13. += and -= are _____ assignment operators
16. The second part of a variable declaration
17. Where objects live
19. A variable declared directly in a class that all its members can access
20. What floats in a float
22. Tells a method to stop immediately, possibly sending a value back to the statement that called it
24. The statement you use to create an object
25. If a method's return type is _____, it doesn't return anything

Bullet Points

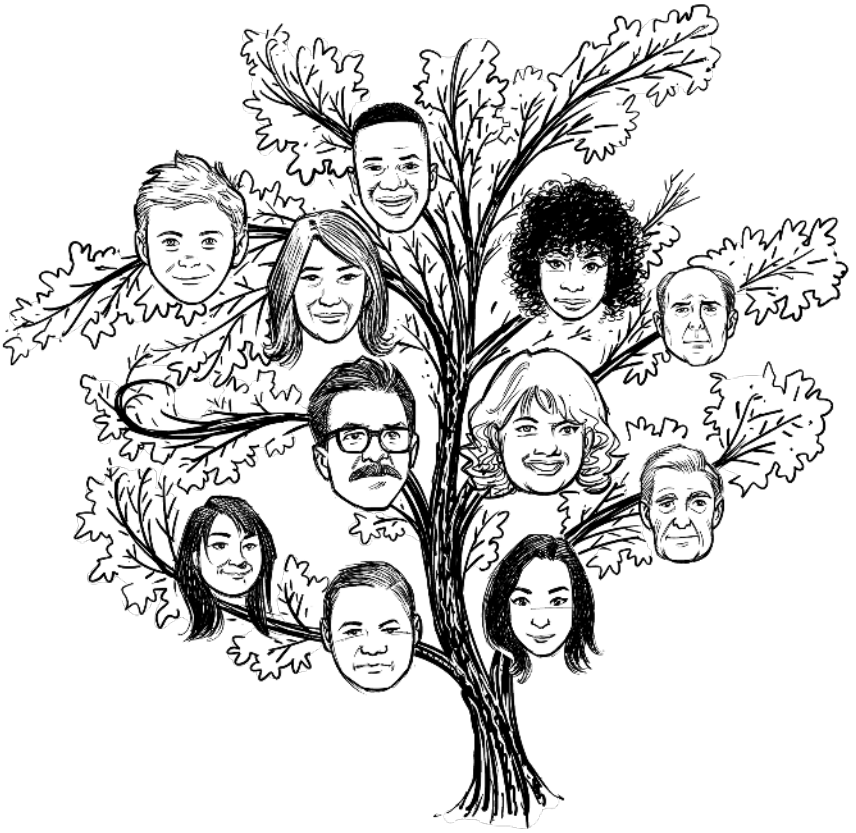
- **Encapsulation** keeps your code safe by preventing classes from modifying it unexpectedly or otherwise misusing other classes' members.
- Fields that require some processing or calculation to happen when they're set are **prime candidates** for encapsulation.
- Think about ways fields and methods can be **misused**. Only make fields and methods public if you need to.
- Using consistent case when choosing names for fields, properties, variables, and methods makes code easier to read. Many developers use **camelCase** for private fields and **PascalCase** for public ones.
- A **property** is a class member that looks like a field when it's used, but acts like a method when it runs.
- A **get accessor** (or **getter**) is defined by the `get` keyword followed by a method that returns the value of the property.
- A **set accessor** (or **setter**) is defined by the `set` keyword and followed by a method that sets the value of the property. Inside the method, the `value` keyword is a read-only variable that contains the value being set.
- Properties often get or set a **backing field**, or a private field that's encapsulated by restricting access to it through a property.
- An **auto-implemented property**—sometimes called an **automatic property** or **auto-property**—is a property that has a getter that returns the value of the backing field, and a setter that updates it.
- Use the **prop snippet** in Visual Studio to create an auto-implemented property by typing **`prop`** followed by two tabs.
- Use the **`private` keyword** to restrict access to a get or set accessor. A read-only property has a private set accessor.
- When an object is created, the CLR first **sets** all of the fields that have values set in their declarations and then **executes** the constructor, before **returning** to the `new` statement that created the object.
- Use a **constructor with parameters** to initialize properties. Specify arguments to pass to the constructor when you use the `new` keyword.
- A parameter with the same name as a field **masks** that field. Use the **`this` keyword** to access the field.
- If you don't add a constructor to your class, the C# compiler automatically adds a **parameterless constructor** for you.
- You can keep a class from being instantiated by other classes by adding a **private constructor**.

Objectcross solution



6 inheritance

Your object's family tree



Sometimes you DO want to be just like your parents.

Ever run across a class that ***almost*** does exactly what you want ***your*** class to do? Found yourself thinking that if you could just ***change a few things***, that class would be perfect? With **inheritance**, you can **extend** an existing class so your new class gets all of its behavior—with the **flexibility** to make changes to that behavior so you can tailor it however you want. Inheritance is one of the most powerful concepts and techniques in the C# language: with it, you can **avoid duplicate code**, **model the real world** more closely, and end up with apps that are **easier to maintain** and **less prone to bugs**.

Calculate damage for MORE weapons

Do this!

The updated sword damage calculator was a huge hit on game night! Now Owen wants calculators for all of the weapons. Let's start with the damage calculation for an arrow, which uses a 1d6 roll. Let's **create a new ArrowDamage class** to calculate the arrow damage using the arrow formula in Owen's game master notebook.

- THE BASE DAMAGE FOR AN ARROW IS THE 1D6 ROLL MULTIPLIED BY .35HP.
- FOR A MAGIC ARROW, THE BASE DAMAGE IS MULTIPLIED BY 2.5HP.
- A FLAMING ARROW ADDS AN EXTRA 1.25HP.
- THE RESULT IS ROUNDED **UP** TO THE NEAREST INTEGER HP.

Most of the code in ArrowDamage will be ***identical to the code*** in the SwordDamage class. Here's what we need to do to get started building the new app.

1. **Create a new .NET Console App project.** We want it to do both sword and arrow calculations, so **add the SwordDamage class** to the project. The constants aren't used anywhere else in the app, so **make them private**.

2. **Create an ArrowDamage class that's an *exact copy* of SwordDamage.** Create a new class called ArrowDamage, then **copy all of the code from SwordDamage and paste it** into the new ArrowDamage class. Then change the constructor name to ArrowDamage so the program builds.

ArrowDamage
Roll Magic Flaming Damage

3. **Refactor the constants.** The arrow damage formula has different values for the base and flame damage, so let's rename the BASE_DAMAGE constant to BASE_MULTIPLIER and update the constant values. We think these constants make the code easier to read, so add a MAGIC_MULTIPLIER constant too:

```
private const decimal BASE_MULTIPLIER = 0.35M;
private const decimal MAGIC_MULTIPLIER = 2.5M;
private const decimal FLAME_DAMAGE = 1.25M;
```

Do you agree with us that these constants make the code easier to read? It's OK if you don't!

4. **Modify the CalculateDamage method.** Now all you need to do to make your new ArrowDamage class work is to update the CalculateDamage method so it does the correct calculation:

```
private void CalculateDamage()
{
    decimal baseDamage = Roll * BASE_MULTIPLIER;
    if (Magic) baseDamage *= MAGIC_MULTIPLIER;
    if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
    else Damage = (int) Math.Ceiling(baseDamage);
}
```

You can use the *Math.Ceiling method* to round values up. It keeps the type, so you still need to cast to an int.

Brain Power

There are **many** different ways to write code that does the same thing. Can you think of another way to write the code to calculate arrow damage?

Use a switch statement to match several candidates

Let's update our console app to prompt the user whether to calculate damage from an arrow or a sword. We'll ask for a key, and use the static **Char.ToUpper method** to convert it to uppercase:

```
Console.Write("\nS for sword, A for arrow, anything else to quit: ");
char weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);
```

The Char.ToUpper method converts 's' and 'a' to 'S' and 'A.'

We ***could*** use `if/else` statements for this:

```
if (weaponKey == 'S') { /* calculate sword damage */ }
else if (weaponKey == 'A') { /* calculate arrow damage */ }
else return;
```

That's how we've handled input so far. Comparing one variable against many different values is a really common pattern that you'll see over and over again. It's so common that C# has a special kind of statement *designed specifically for this situation*. A **`switch` statement** lets you compare one variable against many values in a way that's compact and easy to read. Here's a `switch` statement that does exactly the same thing as the previous `if/else` statements:

```
switch (weaponKey)
{
    case 'S':
        /* calculate sword damage */
        break;
    case 'A':
        /* calculate arrow damage */
        break;
    default:
        return;
}
```

You'll start with the `switch` keyword, followed by whatever is going to be compared against a bunch of different possible values.

The body of the `switch` statement is a series of cases that compare whatever follows the `switch` keyword against a particular value. Each case begins with the `case` keyword, the value to match, and a colon, and ends with a `break` statement to signal the end of the case.

The `default` keyword is like a final `else` statement at the end of a series of `if/else` statements. This is what the `switch` does if none of the cases matched.

Exercise

Update the top-level statements to use a `switch` statement to let the user choose the type of weapon. Start by copying the top-level statements from the exercise solution at the end of Chapter 5.

1. Create an instance of ArrowDamage at the top of the method, just after you create the SwordDamage instance.
2. Modify the RollDice method to take an int parameter called `numberOfRolls` so you can call RollDice(3) to roll 3d6 (so it calls Random.Shared.Next(1, 7) three times and adds the results), or RollDice(1) to roll 1d6.
3. Add the two lines of code exactly like they appear above that write the sword or arrow prompt to the console, read the input using Console.ReadKey, use Char.ToUpper to convert the key to uppercase, and store it in weaponKey.
4. **Add the `switch` statement.** It will be exactly the same as the preceding `switch` statement, except you'll replace each of the /* comments */ with code that calculates damage and writes a line of output to the console.

Exercise Solution

We just gave you a totally new piece of C# syntax—the **`switch` statement**—and asked you to use it in a program. The C# team at Microsoft is constantly improving the language, and being able to incorporate new language elements into your code is **a really valuable C# skill**. We designed this exercise to help give you some practice with it.

```
SwordDamage swordDamage = new SwordDamage(RollDice(3));
ArrowDamage arrowDamage = new ArrowDamage(RollDice(1));

while (true)
{
    Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                  "3 for both, anything else to quit: ");
    char key = Console.ReadKey().KeyChar;
    if (key != '0' && key != '1' && key != '2' && key != '3') return;

    Console.Write("\nS for sword, A for arrow, anything else to quit: ");
    char weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);

    switch (weaponKey)
    {
        case 'S':
            swordDamage.Roll = RollDice(3);
            swordDamage.Magic = (key == '1' || key == '3');
            swordDamage.Flaming = (key == '2' || key == '3');
            Console.WriteLine(
               $"\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\n");
            break;

        case 'A':
            arrowDamage.Roll = RollDice(1);
            arrowDamage.Magic = (key == '1' || key == '3');
            arrowDamage.Flaming = (key == '2' || key == '3');
            Console.WriteLine(
               $"\nRolled {arrowDamage.Roll} for {arrowDamage.Damage} HP\n");
            break;

        default:
            return;
    }
}

int RollDice(int numberOfRolls)
{
    int total = 0;
    for (int i = 0; i < numberOfRolls; i++) total += Random.Shared.Next(1, 7);
    return total;
}
```

Create an instance of the new ArrowDamage class that you added.

This block of code is almost identical to the program from Chapter 5. Instead of using it in an if/else block, it's in a case in a switch statement (and it passes an argument to RollDice).

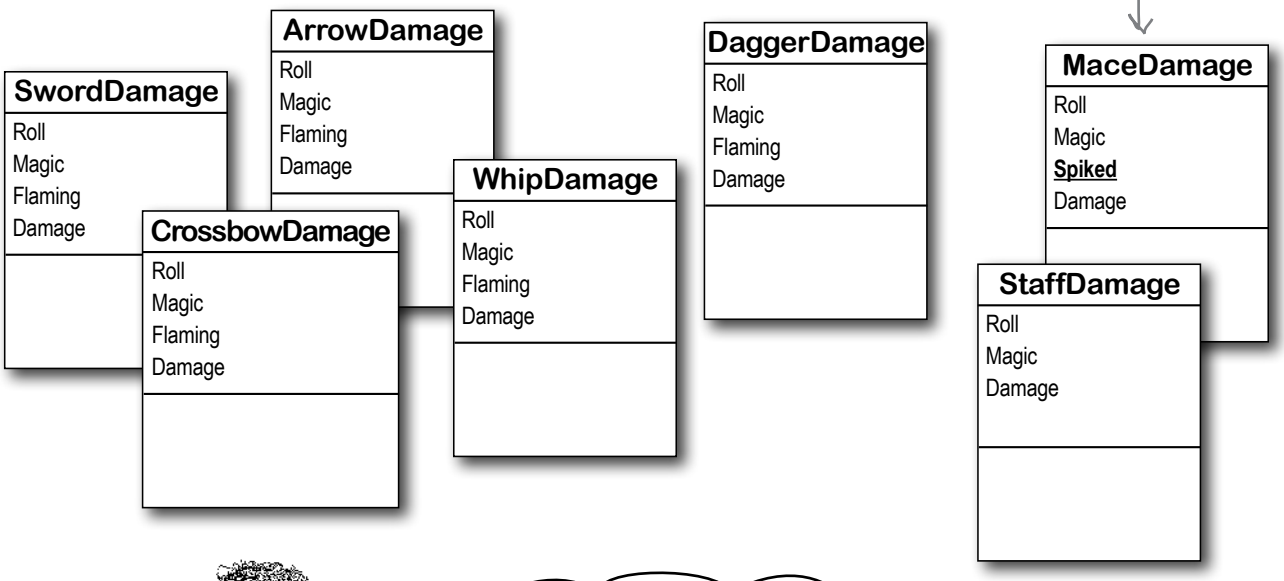
The code to use the instance of ArrowDamage to calculate damage is really similar to the code for SwordDamage. In fact, it's *almost identical*. Is there a way that we can reduce the duplication of code and make our program easier to read?

Try this! Set a breakpoint on switch (weaponKey), then use the debugger to step through the `switch` statement. That's a great way to really get a sense of how it works. Then try removing one of the `break` lines and stepping through it—the execution continues (or *falls through*) to the next case.

One more thing...can we calculate damage for a dagger? And a mace? And a staff? and...

We've made two classes for sword and arrow damage. But what happens if there are three other weapons? Or four? Or 12? And what if you had to maintain that code and make more changes later? What if you had to make the same exact change to five or six closely related classes? What if you had to keep making changes? It's inevitable that bugs would slip through—it's way too easy to update five classes but forget to change the sixth.

What if some of the classes are related, but not quite identical? What if a mace can be spiked or not, but it can't be flaming? Or if a staff can't be either of those things?



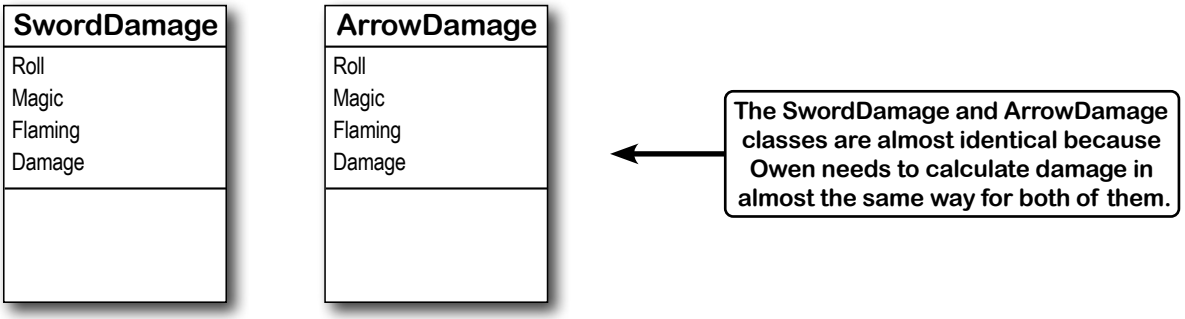
Wow, I'd have to write the same code over and over again. That's a ***really inefficient way to work****. There's got to be a better way.*

You're right! Having the same code repeated in different classes is inefficient and error-prone.

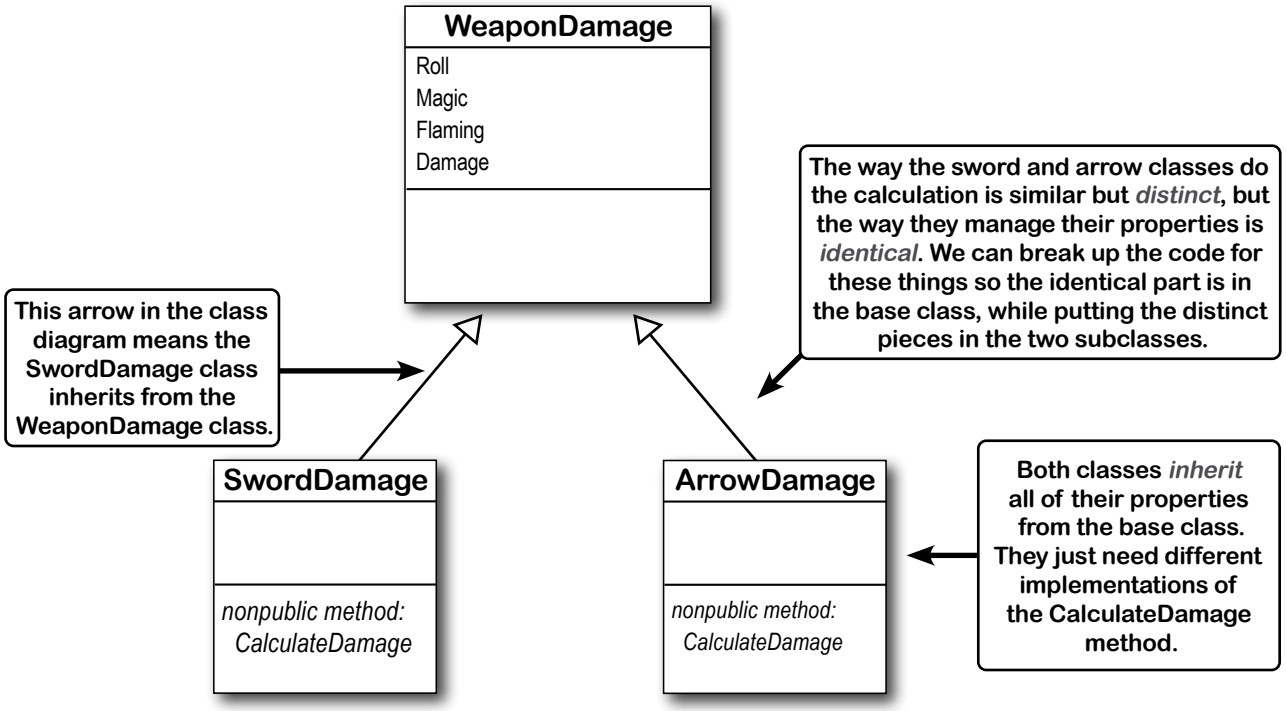
Lucky for us, C# gives us a better way to build classes that are related to each other and share behavior: ***inheritance***.

When your classes use inheritance, you only need to write your code once

It's no coincidence that your SwordDamage and ArrowDamage classes have a lot of the same code. When you write C# programs, you often create classes that represent things in the real world, and those things are usually related to each other. Your classes have **similar code** because the things they represent in the real world—two similar calculations from the same role-playing game—have **similar behaviors**.



When you have two classes that are specific cases of something more general, you can set them up to **inherit** from the same class. When you do that, each of them is a **subclass** of the same **base class**.



Build up your class model by starting general and getting more specific

When you build a set of classes that represent things (especially things in the real world), you're building a **class model**. Real-world things are often in a **hierarchy** that goes from more general to more specific, and your programs have their own **class hierarchy** that does the same thing. In your class model, classes further down in the hierarchy **inherit** from those above them.

General

In a class model, the Cheese class might inherit from DairyProduct, which would inherit from Food.

Food

Dairy Product

Cheese

Cheddar

Aged Vermont Cheddar

Specific

If you have a recipe that calls for cheddar cheese, then you can use aged Vermont cheddar. If it specifically needs aged Vermont, then you can't just use any cheddar. You need that *specific* cheese.

General

Every bird is an animal, but not every animal is a bird.

Animal

Bird

To someone looking for a pet, any songbird might do. To an ornithologist studying the Mimidae bird family, confusing the northern and southern mockingbirds would be unacceptable.

Songbird

Mockingbird

Northern Mockingbird

Specific

Something lower in the hierarchy inherits most or all of the attributes of everything above it. All animals eat and mate, so northern mockingbirds eat and mate.

in-her-it, verb.
to derive an attribute from one's parents or ancestors. *She wanted the baby to* ***inherit*** *her big brown eyes, and not her husband's beady blue ones.*

How would you design a zoo simulator?

Lions and tigers and bears…oh my! Also, hippos, wolves, and the occasional dog. Your job is to design an app that simulates a zoo. (Don't get too excited—we're not going to actually build the code, just design the classes to represent the animals. We bet you're already thinking about how you'd do this in Unity!)

We've been given a list of some of the animals that will be in the program, but not all of them. We know that each animal will be represented by an object, and that the objects will move around in the simulator, doing whatever it is that each particular animal is programmed to do.

More importantly, we want the program to be easy for other programmers to maintain, which means they'll need to be able to add their own classes later on if they want to add new animals to the simulator.

Let's start by building a class model for the animals we know about.

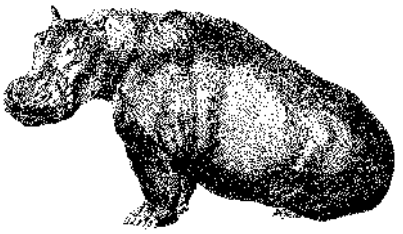
So what's the first step? Well, before we can talk about **specific** animals, we need to figure out the **general** things they have in common—the abstract characteristics that all animals have. Then we can build those characteristics into a base class that all animal classes can inherit from.

The terms *parent*, *superclass*, and *base class* are often used interchangeably. Also, the terms *extend* and *inherit from* mean the same thing. The terms *child* and *subclass* are also synonymous, but subclass can also be used as a verb.

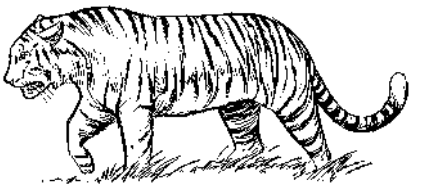
Some people use the term "base class" to specifically mean the class at the top of the inheritance tree...but not the VERY top, because every class inherits from Object or a subclass of Object.

1 **Look for things the animals have in common.**

Take a look at these six animals. What do a lion, a hippo, a tiger, a bobcat, a wolf, and a dog have in common? How are they related? You'll need to figure out their relationships so you can come up with a class model that includes all of them.

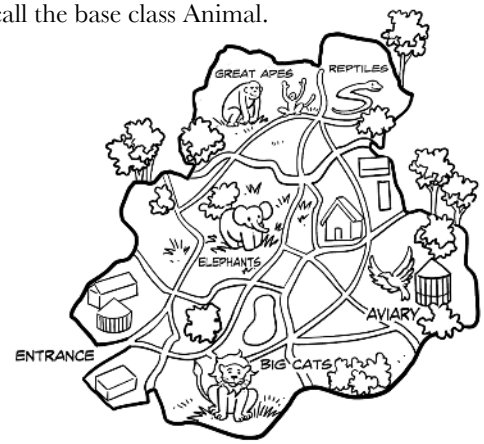


The zoo simulator includes a guard dog that roams the grounds protecting the animals.



2 Build a base class to give the animals everything they have in common.

The fields, properties, and methods in the base class will give all of the animals that inherit from it a common state and behavior. They're all animals, so it makes sense to call the base class Animal.

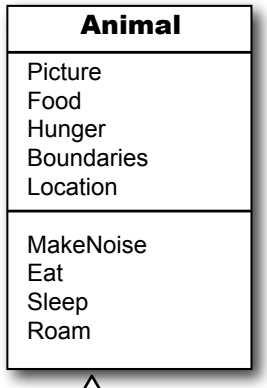


You already know that we should avoid duplicate code: it's hard to maintain, and always leads to headaches down the road. So let's choose fields and methods for an Animal base class that you **only have to write once**, and each of the animal subclasses can inherit them. Let's start with the public properties:

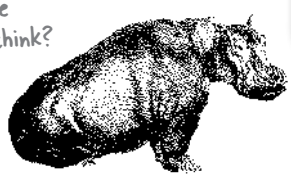
- Picture: a path to an image file.
- Food: the type of food this animal eats. Right now, there can be only two values: meat and grass.
- Hunger: an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.
- Boundaries: a reference to a class that stores the height, width, and location of the pen that the animal will roam around in.
- Location: the X and Y coordinates where the animal is standing.

In addition, the Animal class has four methods the animals can inherit:

- MakeNoise: a method to let the animal make a sound
- Eat: behavior for when the animal encounters its preferred food
- Sleep: a method to make the animal lie down and take a nap
- Roam: a method to make animals wander around their pens



Choosing a base class is about making choices. You could have decided to use a ZooOccupant class that defines the feed and maintenance costs, or an Attraction class with methods for how the animals entertain the zoo visitors. We think Animal makes the most sense here. What do you think?



Different animals have different behaviors

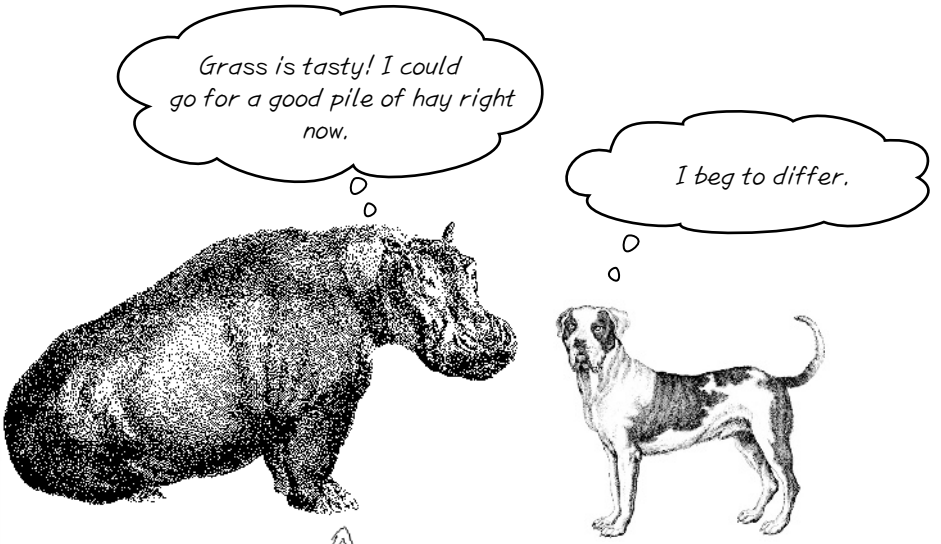
Lions roar, dogs bark, and hippos, as it turns out, make a wide variety of sounds to tell each other all sorts of things. All of the classes that inherit from Animal will have a MakeNoise method, but each of those methods will work a different way and will have different code. When a subclass changes the behavior of one of the methods that it inherited, we say that it **overrides** the method.

Just because a property or a method is in the Animal base class, that doesn't mean every subclass has to use it the same way...or at all!

3 **Figure out what each animal does that the Animal class does differently—or not at all.**

Every animal needs to eat, but a dog might take small bites of meat while a hippo eats huge mouthfuls of grass. What would the code for that behavior look like? Both the dog and the hippo would override the Eat method. The hippo's method would have it consume, say, 20 pounds of hay each time it was called. The dog's Eat method, on the other hand, would reduce the zoo's food supply by one 12-ounce can of dog food.

So when you've got a subclass that inherits from a base class, it must inherit all of the base class's behaviors... but you can modify them in the subclass so they're not performed exactly the same way. That's what overriding is all about.



Animal
Picture Food Hunger Boundaries Location
MakeNoise Eat Sleep Roam

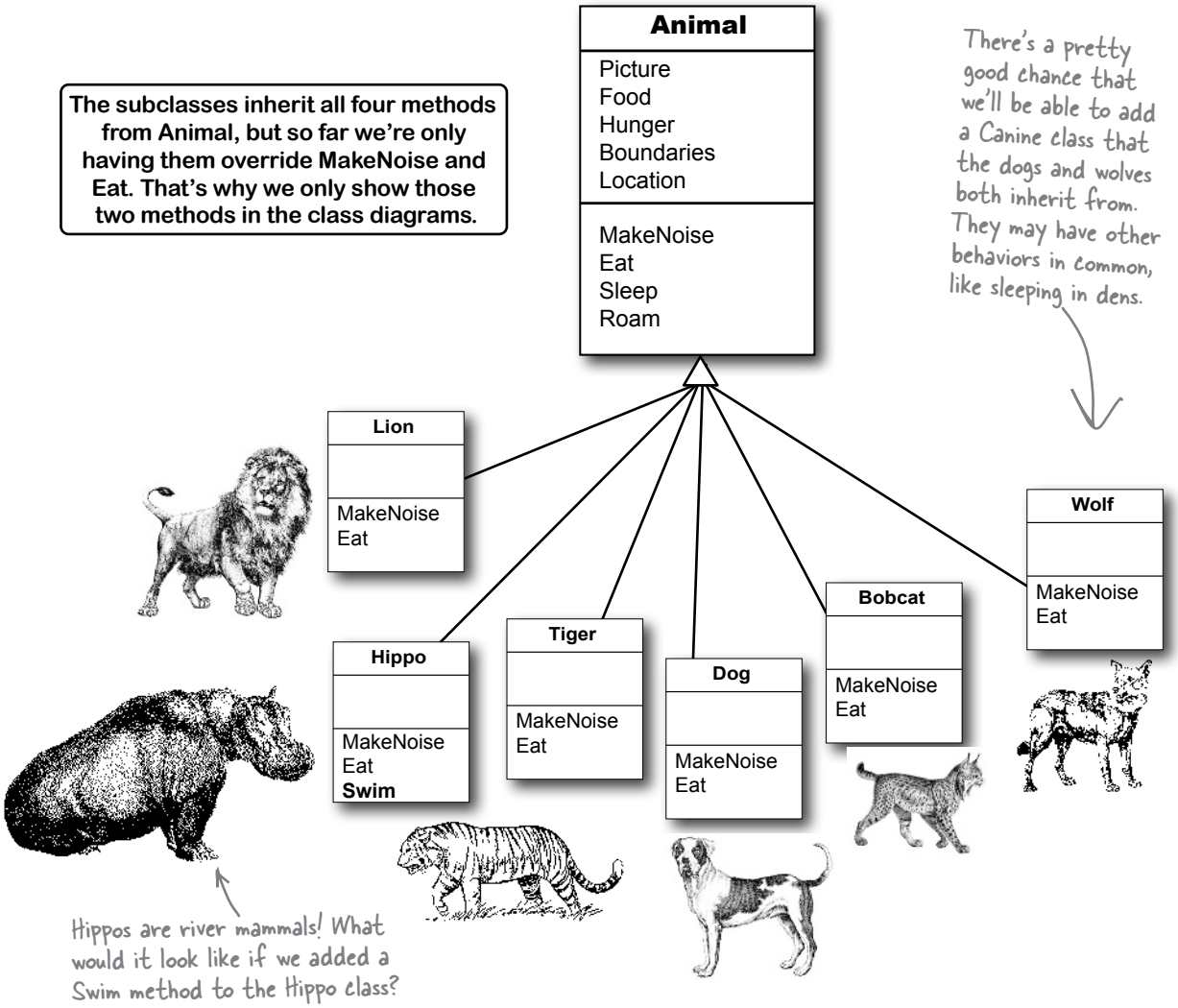
Brain Power

We already know that some animals will override the MakeNoise and Eat methods. Which animals will override Sleep or Roam? Will any of them?

4 Look for classes that have a lot in common.

Don't dogs and wolves seem pretty similar? They're both canines, and it's a good bet that if you look at their behavior they have a lot in common. They probably eat the same food and sleep the same way. What about bobcats, tigers, and lions? It turns out all three of them move around their habitats in exactly the same way. It's a good bet that you'll be able to have a general Feline class that lives between Animal and those three feline classes that can help prevent duplicate code between them.

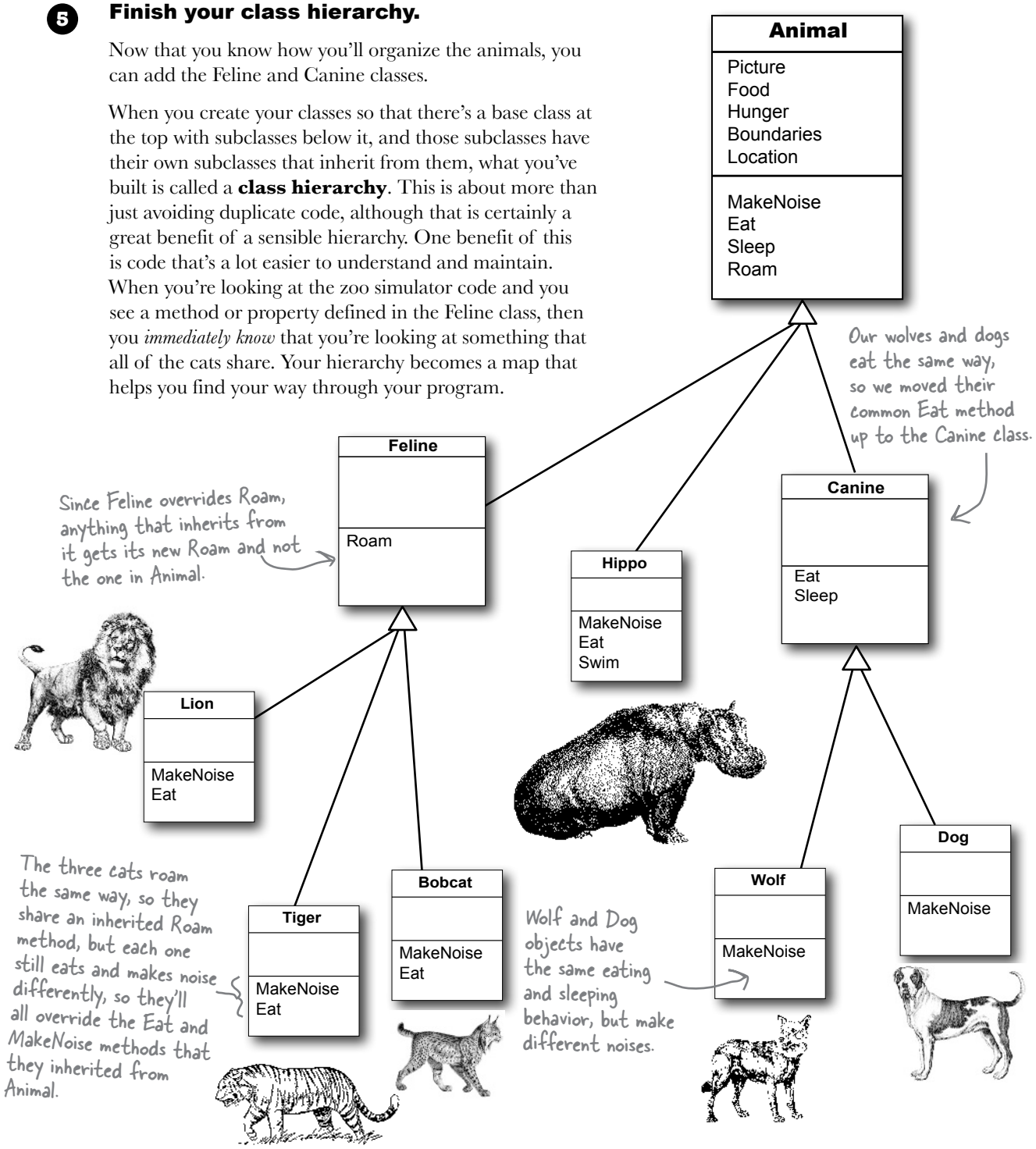
The subclasses inherit all four methods from Animal, but so far we're only having them override MakeNoise and Eat. That's why we only show those two methods in the class diagrams.



5 Finish your class hierarchy.

Now that you know how you'll organize the animals, you can add the Feline and Canine classes.

When you create your classes so that there's a base class at the top with subclasses below it, and those subclasses have their own subclasses that inherit from them, what you've built is called a **class hierarchy**. This is about more than just avoiding duplicate code, although that is certainly a great benefit of a sensible hierarchy. One benefit of this is code that's a lot easier to understand and maintain. When you're looking at the zoo simulator code and you see a method or property defined in the Feline class, then you *immediately know* that you're looking at something that all of the cats share. Your hierarchy becomes a map that helps you find your way through your program.

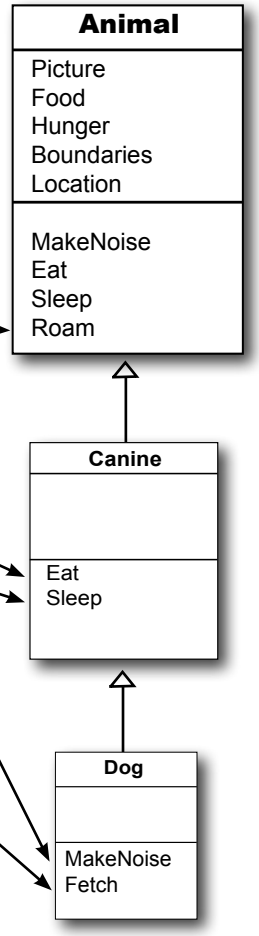


Every subclass extends its base class

You're not limited to the methods that a subclass inherits from its base class…but you already know that! After all, you've been building your own classes all along. When you modify a class to make it inherit members—and we'll see that in C# code soon!—what you're doing is taking the class you've already built and ***extending*** it by adding all of the fields, properties, and methods in the base class. So if you want to add a Fetch method to Dog, that's perfectly normal. It won't inherit or override anything—only the Dog class will have that method, and it won't end up in Wolf, Canine, Animal, Hippo, or any other class.

hi-er-ar-chy, noun.
an arrangement or classification in which groups or things are ranked one above the other. *The president of Dynamco had worked her way up from the mailroom to the top of the corporate* ***hierarchy***.

MAKES A NEW INSTANCE OF DOG	`Dog spot = new Dog();`
CALLS THE VERSION IN DOG	`spot.MakeNoise();`
CALLS THE VERSION IN ANIMAL	`spot.Roam();`
CALLS THE VERSION IN CANINE	`spot.Eat();`
CALLS THE VERSION IN CANINE	`spot.Sleep();`
CALLS THE VERSION IN DOG	`spot.Fetch();`

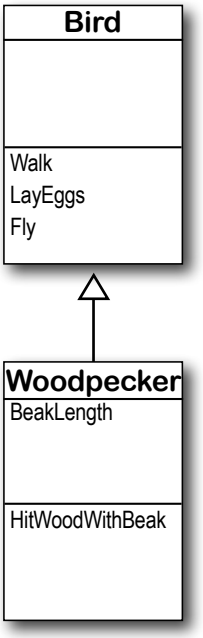


C# always calls the most specific method

If you tell your Dog object to roam, there's only one method that can be called—the one in the Animal class. What about telling your Dog to make noise? Which MakeNoise is called?

Well, it's not too hard to figure it out. A method in the Dog class tells you how dogs make noise. If it's in the Canine class, it's telling you how all canines do it. If it's in Animal, then it's a description of that behavior that's so general that it applies to every single animal. So if you ask your Dog to make a noise, first C# will look inside the Dog class to find the behavior that applies specifically to dogs. If Dog didn't have a MakeNoise method, it'd check Canine, and after that it'd check Animal.

Any place where you can use a base class, you can use one of its subclasses instead



One of the most useful things you can do with inheritance is to **extend** a class. So if your method takes a Bird object, then you can pass an instance of Woodpecker. All that method knows is that it has a bird. It doesn't know what kind of bird it has, so it can only ask it to do things that all birds do: it can ask the bird to Walk and LayEggs, but it can't ask it to HitWoodWithBeak, because only Woodpeckers have that behavior—and the method doesn't know that it's specifically a Woodpecker, just that it's a more general Bird. It ***only has access to the fields, properties, and other methods that are part of the class it knows about***.

Let's see how this works in code. Here's a method that takes a Bird reference:

```
public void IncubateEggs(Bird bird)
{
    bird.Walk(incubatorEntrance);
    Egg[] eggs = bird.LayEggs();
    AddEggsToHeatingArea(eggs);
    bird.Walk(incubatorExit);
}
```

Even if we pass a Woodpecker object to IncubateEggs, it's a Bird reference, so we can only use Bird class members.

If you want to incubate some Woodpecker eggs, you can pass a Woodpecker reference to the IncubateEggs method, because a Woodpecker is a *kind of* Bird—which is why it inherits from the Bird class:

```
public void GetWoodpeckerEggs()
{
    Woodpecker woody = new Woodpecker();
    IncubateEggs(woody);
    woody.HitWoodWithBeak();
}
```

You can ***replace a superclass with a subclass***, but you can't replace a subclass with its superclass. You can pass a Woodpecker to a method that takes a Bird reference, but not vice versa:

This should make intuitive sense. If someone asks you for a bird and you hand them a woodpecker, they'll be happy. But if they ask you for a woodpecker and you hand them a pigeon, they'll be confused.

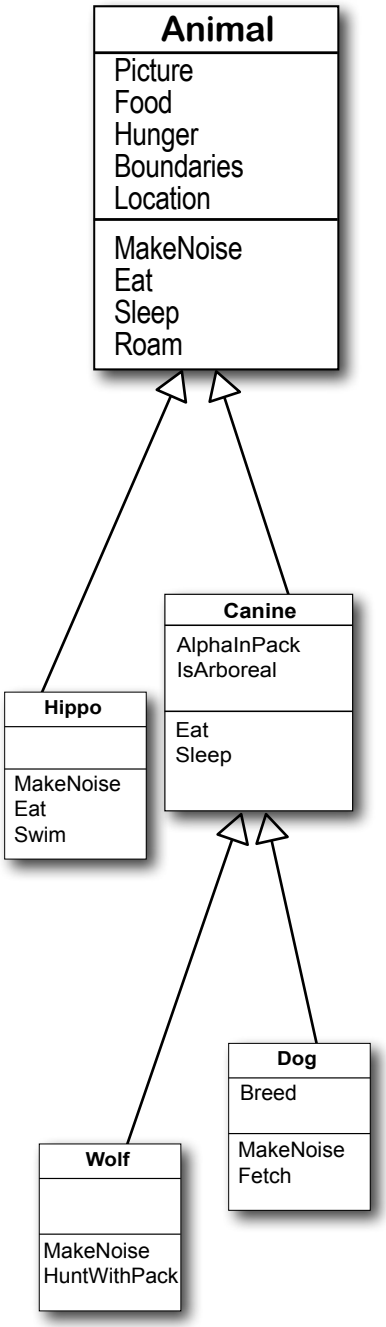
```
public void GetWoodpeckerEggs_Take_Two()
{
    Woodpecker woody = new Woodpecker();
    woody.HitWoodWithBeak();

    // This line copies the Woodpecker reference to a Bird variable
    Bird birdReference = woody;
    IncubateEggs(birdReference);

    // THE NEXT LINE WILL HAVE A COMPILER ERROR!!!
    Woodpecker secondWoodyReference = birdReference;

    secondWoodyReference.HitWoodWithBeak();
}
```

You can assign woody to a Bird variable because a woodpecker is a kind of bird...

...but you can't assign birdReference back to a Woodpecker variable, because not every bird is a woodpecker! That's why this line will cause an error.

Sharpen your pencil

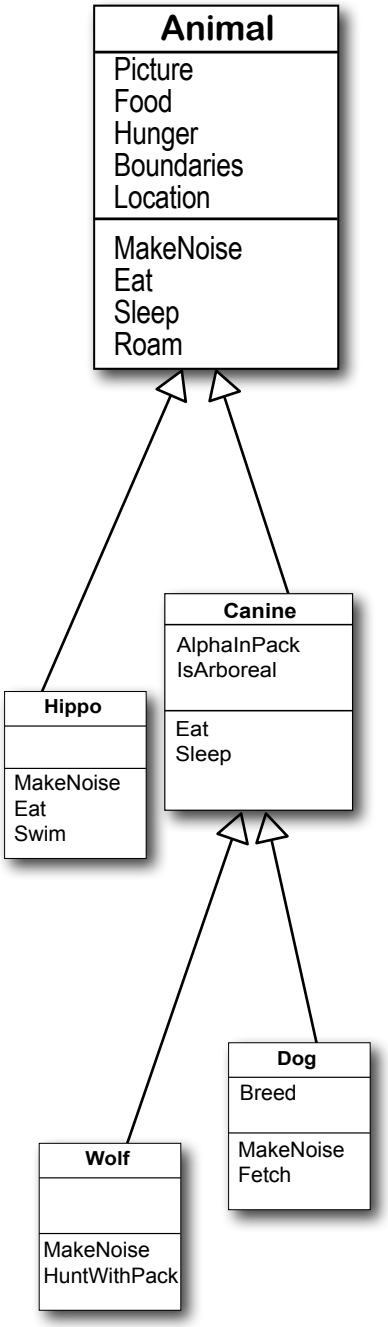
The code below is from a program that uses the class model that includes Animal, Hippo, Canine, Wolf, and Dog. Draw a line through each statement that won't compile, and write an explanation for the problem next to it.




```
Canine canis = new Dog();
Wolf charon = new Canine();
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();

Dog fido = canis;
Animal visitorPet = fido;
Animal harvey = bailey;
harvey.Roam();
harvey.Swim();
harvey.Sleep();
harvey.Eat();

Hippo brutus = harvey;
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();

Canine london = new Wolf();
Wolf egypt = london;
egypt.HuntWithPack();
egypt.HuntWithPack();
egypt.AlphaInPack = false;
Dog rex = london;
rex.Fetch();
```

Sharpen your pencil Solution

The code below is from a program that uses the class model that includes Animal, Hippo, Canine, Wolf, and Dog. Draw a line through each statement that won't compile, and write an explanation for the problem next to it.




```
Canine canis = new Dog();
Wolf charon = new Canine();
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();
```

Wolf is a subclass of Canine, so you can't assign a Canine object to a Wolf. Think of it this way: a wolf is a type of canine, but not every canine is a wolf.

Even though the canis variable is a reference to a Dog object, the type of the variable is Canine, so you can't assign it to a Dog.

```
Dog fido = canis;
Animal visitorPet = fido;
Animal harvey = bailey;
harvey.Roam();
harvey.Swim();
harvey.Sleep();
harvey.Eat();
```

harvey is a reference to a Hippo object, but the harvey variable is an Animal, so you can't use it to call the Hippo.Swim method.

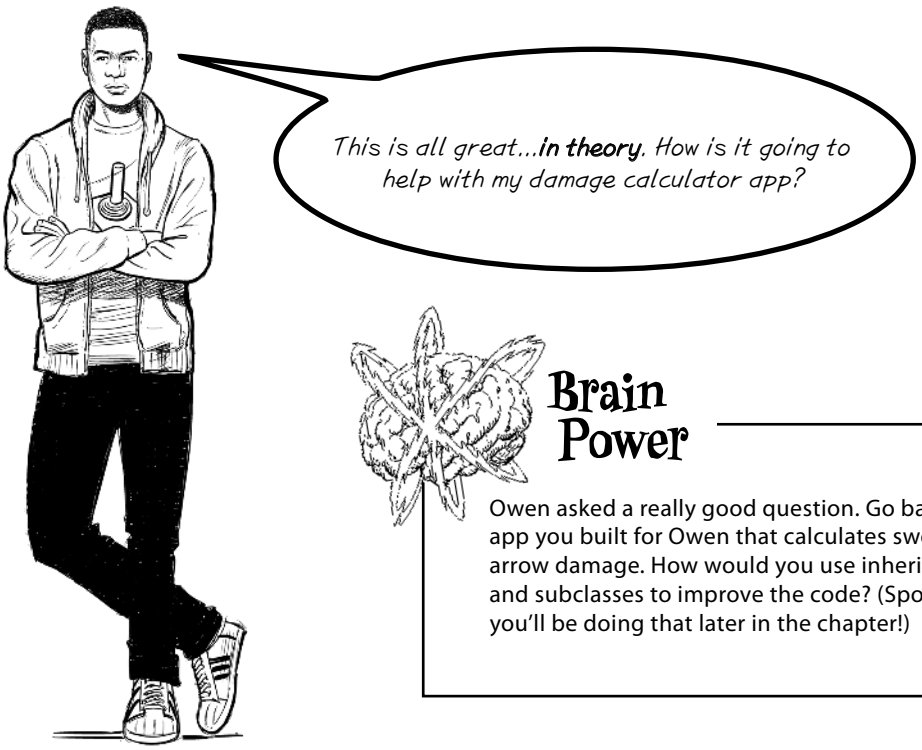
```
Hippo brutus = harvey;
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();
```

This doesn't work for the same reason that Dog fido = canis; didn't work. harvey may point to a Hippo object, but its type is Animal, and you can't assign an Animal to a Hippo variable.

```
Canine london = new Wolf();
Wolf egypt = london;
egypt.HuntWithPack();
egypt.HuntWithPack();
egypt.AlphaInPack = false;
Dog rex = london;
rex.Fetch();
```

This is the same problem! You can assign a Wolf to a Canine, but you can't assign a Canine to a Wolf...

...and you definitely can't assign a Wolf to a Dog.



Brain Power

Owen asked a really good question. Go back to the app you built for Owen that calculates sword and arrow damage. How would you use inheritance and subclasses to improve the code? (Spoiler alert: you'll be doing that later in the chapter!)

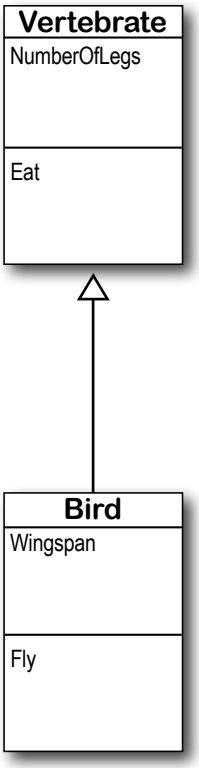
Bullet Points

- A **`switch` statement** lets you compare one variable against many values. Each case executes code if its value matches. The default block runs if no cases match.
- **Inheritance** lets you build classes that are related to each other and share behavior. Use arrows to show inheritance in a class diagram.
- When you have two classes that are **specific** cases of something more **general**, you can set them up to inherit from the same general class. When you do that, each of them is a **subclass** of the same general **base class**.
- When you build a set of classes that represent things, it's called a **class model**. It can include classes that form a **hierarchy** of subclasses and a base class.
- The terms **parent**, **superclass**, and **base class** are often used interchangeably. Also, the terms **extend** and **inherit from** mean the same thing.
- The terms **child** and **subclass** mean the same thing. We say that a subclass **extends** its base class. (The word **subclass** can also be used as a verb.)
- When a subclass changes the behavior of one of the methods that it inherited, we say that it **overrides** the method.
- C# always calls the **most specific method**. If a method in the base class uses a method or property that the subclass overrides, it will call the overridden version in the subclass.
- You can **use a subclass reference** in place of a base class. If a method takes an `Animal` parameter and Dog extends Animal, you can pass it a Dog argument.
- You can always use a subclass **in place of the base class** it inherits from, but you can't always use a base class in place of a subclass that extends it.

Use a colon to extend a base class

When you're writing a class, you use a **colon (:)** to have it inherit from a base class. That makes it a subclass, and gives it **all of the fields, properties, and methods** of the class it inherits from. This Bird class is a subclass of Vertebrate:

When a subclass extends a base class, it inherits its members: all of the fields, properties, and methods in the base class. They're automatically added to the subclass.




```
class Vertebrate
{
   public int Legs { get; set; }

   public string Eat() {
     // code to make it eat
   }
}
```

The Bird class uses a colon to inherit from the Vertebrate class. This means that it inherits all of the fields, properties, and methods from Vertebrate.

```
class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
      // code to make a bird fly
    }
}
```

The base class follows the colon in the class declaration. In this case, Bird extends Vertebrate.

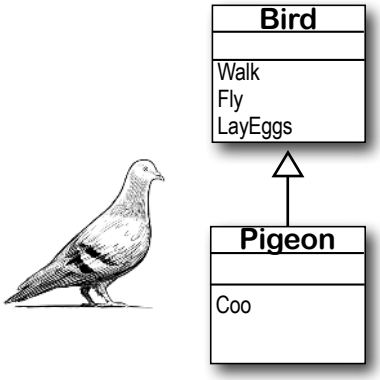
```
Bird tweety = new Bird();
Console.WriteLine(tweety.Wingspan);
tweety.Fly();
tweety.Legs = 2;
Console.Write(tweety.Eat());
```

tweety is an instance of Bird, so it has the Bird methods, properties, and fields as usual.

Since the Bird class extends Vertebrate, every instance of Bird also has the members defined in the Vertebrate class.

We know that inheritance adds the base class fields, properties, and methods to the subclass...

We've seen inheritance when a subclass needs to inherit all of the base class's methods, properties, and fields.

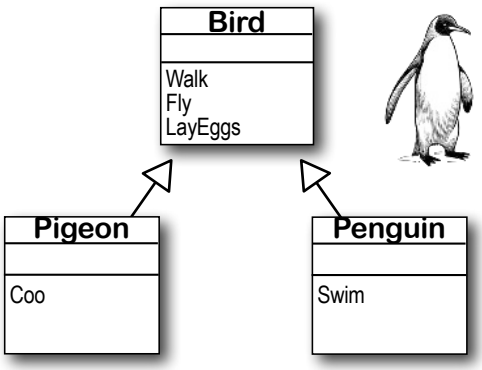



```
class Bird {
   public void Fly() {
       /* code to make birds fly */
   }
   public void LayEggs() { ... };
   public void PreenFeathers() { ... };
}


class Pigeon : Bird {
   public void Coo() { ... }
}

public void SimulatePigeon() {
    Pigeon Harriet = new Pigeon();

    // Since Pigeon is a subclass of Bird,
    // we can call methods from either class.
    Harriet.Walk();
    Harriet.LayEggs();
    Harriet.Coo();
    Harriet.Fly();
}

class Penguin : Bird {
   public void Swim() { ... }
}

public void SimulatePenguin() {
    Penguin Izzy = new Penguin();
    Izzy.Walk();
    Izzy.LayEggs();
    Izzy.Swim();
    Izzy.Fly();
}
```

This code will compile because Penguin extends Bird. Is there a way to change the Penguin class so it displays a warning if a penguin tries to fly?

...but some birds don't fly!

What do you do if your base class has a method that your subclass needs to modify?



Oops—we've got a problem. Penguins are birds, and the Bird class has a Fly method, but we don't want our penguins to fly. It would be great if we could display a warning if a penguin tries to fly.

Brain Power

If these classes were in your zoo simulator, what would you do about flying penguins?

A subclass can override methods to change or replace members it inherited

Sometimes you've got a subclass that you'd like to inherit *most* of the behaviors from the base class, but *not all of them*. When you want to change the behaviors that a class has inherited, you can **override methods or properties**, replacing them with new members with the same name.

When you **override a method**, your new method needs to have exactly the same signature as the method in the base class it's overriding. In the case of the penguin, that means it needs to be called Fly, return void, and have no parameters.

o-ver-ride, verb.
to use authority to replace, reject, or cancel. *Once Lisa became president of Dynamco, she could* ***override*** *poor management decisions.*

1 **Add the `virtual` keyword to the method in the base class.**
A subclass can only override a method if it's marked with the `virtual` keyword. Adding `virtual` to the Fly method declaration tells C# that a subclass of the Bird class is allowed to override the Fly method.

```
class Bird {
   public virtual void Fly() {
     // code to make the bird fly
   }
}
```

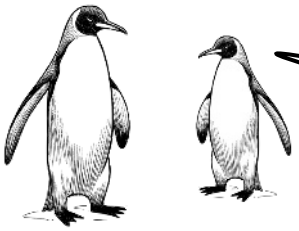
Adding the `virtual` keyword to the Fly method tells C# that a subclass is allowed to override it.

2 **Add the `override` keyword to a method with the same name in the subclass.**
The subclass's method will need to have exactly the same signature—the same return type and parameters—and you'll need to use the `override` keyword in the declaration. Now a Penguin object prints a warning when its Fly method is called.

```
class Penguin : Bird {
   public override void Fly() {
     Console.Error.WriteLine("WARNING");
     Console.Error.WriteLine("Flying Penguin Alert");
   }
}
```

To override the Fly method, add an identical method to the subclass and use the **override** keyword.

We used *Console.Error* to write error messages to the standard error stream (stderr), which is typically used by console apps to print error messages and important diagnostic information.



Keep flapping, Bob, I'm sure we'll get airborne soon!

Exercise Mixed Messages

```
a = 6;
b = 5;
a = 5;
```

```
56
11
65
```

A short C# program is listed here. One block of the program is missing! Your challenge is to match the candidate block of code (on the left) with the output—what's in the message box that the program pops up—that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching output.

Instructions:

1. **Fill in the four blanks in the code.**
2. **Match the code candidates to the output.**

```
class A {
  public int ivar = 7;
  public ___________ string m1() {
    return "A's m1, ";
  }
  public string m2() {
    return "A's m2, ";
  }
  public ___________ string m3() {
    return "A's m3, ";
  }
}

class B : A {
  public ___________ string m1() {
    return "B's m1, ";
  }
}
```

```
class C : B {
  public ___________ string m3() {
    return "C's m3, " + (ivar + 6);
  }
}
```

Here are the top-level statements.

```
A a = new A();
B b = new B();
C c = new C();
A a2 = new C();
string q = "";

[ ]

Console.WriteLine(q);
```

Hint: think really hard about what this line really means.

Candidate code goes here (three lines)

Code candidates:

```
q += b.m1();
q += c.m2();
q += a.m3();
```

```
q += c.m1();
q += c.m2();
q += c.m3();
```

```
q += a.m1();
q += b.m2();
q += c.m3();
```

```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```

Draw a line from each three-line code candidate to the line of output that's produced if you use the candidate in the box.

Lines of output:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, C's m3, 6
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```

(Don't just type this into the IDE—you'll learn a lot more if you figure this out on paper!)

Exercise Solution

Mixed Messages

```
a = 6;
b = 5;
a = 5;
```

```
class A {
  public virtual string m1() {
...
  public virtual string m3() {
}
```

```
class B : A {
  public override string m1() {
...
class C : B {
  public override string m3() {
```

```
q += b.m1();
q += c.m2();
q += a.m3();
```

```
A's m1, A's m2, C's m3, 6
```

```
B's m1, A's m2, A's m3,
```

```
q += c.m1();
q += c.m2();
q += c.m3();
```

```
A's m1, B's m2, C's m3, 6
```

```
B's m1, A's m2, C's m3, 13
```

```
q += a.m1();
q += b.m2();
q += c.m3();
```

```
B's m1, C's m2, A's m3,
```

```
A's m1, B's m2, A's m3,
```

```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```

```
B's m1, A's m2, C's m3, 6
```

```
A's m1, A's m2, C's m3, 13
```

You can always substitute a reference to a subclass in place of a base class because you're using something more specific in place of something more general. So this line:

```
A a2 = new C();
```

means that you're instantiating a new C object, and then creating an A reference called **a2** and pointing it at that object. Names like these make for a good puzzle, but they're pretty hard to understand. Here are a few lines that follow the same pattern, but have names that are more obvious:

```
Canine fido = new Dog();
Bird pidge = new Pigeon();
Feline rex = new Lion();
```

there are no Dumb Questions

Q: A `switch` statement does exactly the same thing as a series of `if/else` statements, right? Isn't that redundant?

A: Not at all. There are many situations where `switch` statements are much more readable than `if/else` statements. For example, let's say you're displaying a menu in a console app, and the user can press a key to choose 1 of 10 different options. What would 10 `if/else` statements in a row look like? We think a `switch` statement would be cleaner and easier to read. You could see at a glance exactly what's being compared, where each option is handled, and what happens in the default case if the user chooses an unsupported option. Also, it's surprisingly easy to accidentally leave off an `else`. If you miss an `else` in the middle of a long string of `if/else` statements, you end up with a really annoying bug that's surprisingly difficult to track down. There are some times when a `switch` statement is easier to read, and other times when `if/else` statements are easier. It's up to you to write code in the way that you think is easiest to understand.

Q: Why does the arrow point up, from the subclass to the base class? Wouldn't the diagram look better with the arrow pointing down instead?

A: It might *seem* more intuitive, but it wouldn't be as accurate. When you set up a class to inherit from another one, you build that relationship into the subclass—the base class remains the same. Its behavior is completely unchanged when you add a class that inherits from it. The base class isn't even aware of this new class. Its methods, fields, and properties remain entirely intact—but the subclass definitely changes its behavior. Every instance of the subclass automatically gets all of the properties, fields, and methods from the base class, and it all happens just by adding a colon. That's why you draw the arrow on your diagram so that it points from the subclass to the base class that it inherits from.

Exercise

Let's get some practice extending a base class. We'll give you the top-level statements for a program that tracks birds laying eggs. Your job is to implement two subclasses of the Bird class.

1. Here are the top-level statements. The app prompts the user for the type of bird and number of eggs to lay:

```
while (true)
{
    Bird bird;
    Console.Write("\nPress P for pigeon, O for ostrich: ");
    char key = Char.ToUpper(Console.ReadKey().KeyChar);
    if (key == 'P') bird = new Pigeon();
    else if (key == 'O') bird = new Ostrich();
    else return;
    Console.Write("\nHow many eggs should it lay? ");
    if (!int.TryParse(Console.ReadLine(), out int numberOfEggs)) return;
    Egg[] eggs = bird.LayEggs(numberOfEggs);
    foreach (Egg egg in eggs)
    {
        Console.WriteLine(egg.Description);
    }
}
```

Here's what the program output looks like:

```
Press P for pigeon, O for ostrich: P
How many eggs should it lay? 4
A 2.40cm white egg
A 1.82cm white egg
A 2.88cm white egg
A 2.28cm white egg

Press P for pigeon, O for ostrich: O
How many eggs should it lay? 3
A 12.48cm speckled egg
A 12.12cm speckled egg
A 12.13cm speckled egg
```

2. Add this Egg class—the constructor sets the size and color:

```
class Egg
{
    public double Size { get; private set; }
    public string Color { get; private set; }
    public Egg(double size, string color)
    {
        Size = size;
        Color = color;
    }
    public string Description {
        get { return $"A {Size:0.00}cm {Color} egg"; }
    }
}
```

Rounding in interpolation

You can round a number to a number of decimal places in string interpolation. We included {Size:0.00} to tell it to round the Size value to two decimal places.

3. This is the Bird class that you'll extend:

```
class Bird
{
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```

4. Create the Pigeon class that extends Bird. Override the LayEggs method and have it lay eggs with the color "white" and a size between 1 and 3 centimeters.

5. Create the Ostrich class that also extends Bird. Override the LayEggs method and have it lay eggs with the color "speckled" and a size between 12 and 13 centimeters.

Exercise Solution

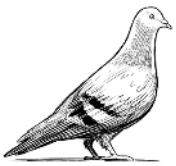
Here are the Pigeon and Ostrich classes. They each have their own version of the LayEggs method that uses the **`override`** keyword in the method declaration. The **`override`** keyword causes the method in the subclass to replace the one that it inherited.

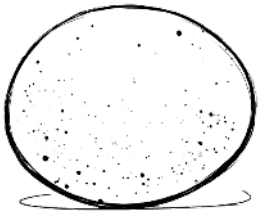
Pigeon is a subclass of Bird, so if you override the LayEggs method, when you create a new Pigeon object and assign it to a Bird variable called **`bird`**, *calling bird.LayEggs* will call the LayEggs method *you defined in Pigeon*.

```
class Pigeon : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Random.Shared.NextDouble() * 2 + 1, "white");
        }
        return eggs;
    }
}
```

The Ostrich subclass works the same way as Pigeon. In both classes, the **`override` keyword** in the LayEggs method declaration means that this new method will replace the LayEggs that it inherited from Bird. So all we need to do is have it create a set of eggs that are the right size and color.

```
class Ostrich : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Random.Shared.NextDouble() + 12, "speckled");
        }
        return eggs;
    }
}
```





Some members are only implemented in a subclass

All the code we've seen so far that works with subclasses has accessed the members from outside the object—like how the top-level statements in the last exercise call LayEggs. Inheritance really shines when the base class **uses a method or property that's implemented in the subclass**. Here's an example. Our zoo simulator has vending machines that let patrons buy soda, candy, and feed to give to the animals in the petting zoo area.

```
class VendingMachine
{
    public virtual string Item { get; }

    protected virtual bool CheckAmount(decimal money) {
        return false;
    }

    public string Dispense(decimal money)
    {
        if (CheckAmount(money)) return Item;
        else return "Please enter the right amount";
    }
}
```

This class uses the `protected` keyword. It's an access modifier that makes a member *public only to its subclasses*, but private to every other class.

VendingMachine is the base class for all vending machines. It has code to dispense items, but those items aren't defined. The method to check if the patron put in the right amount always returns false. Why? Because they **will be implemented in the subclass**. Here's a subclass for dispensing animal feed in the petting zoo:

```
class AnimalFeedVendingMachine : VendingMachine
{
    public override string Item {
        get { return "a handful of animal feed"; }
    }

    protected override bool CheckAmount(decimal money)
    {
        return money >= 1.25M;
    }
}
```

Using the "override" keyword with a property works just as it does when you override a method.

We're using the `protected` keyword for encapsulation. The CheckAmount method is `protected` because it never needs to be called by another class, so only VendingMachine and its subclasses are allowed to access it.

Use the debugger to understand how overriding works

Let's use the debugger to see exactly what happens when we create an instance of AnimalFeedVendingMachine and ask it to dispense some feed. **Create a new Console App project**, then do this.

Debug this!

1. **Add the top-level statements.** Here's the code:

```
VendingMachine vendingMachine = new AnimalFeedVendingMachine();
Console.WriteLine(vendingMachine.Dispense(2.00M));
```

2. **Add the VendingMachine and AnimalFeedVendingMachine classes.** Once they're added, try adding this line of code to the top-level statements:

```
vendingMachine.CheckAmount(1F);
```

You'll get a compiler error because of the `protected` keyword, because only the VendingMachine class or subclasses of it can access its protected methods.

⊗	CS0122	'VendingMachine.CheckAmount(decimal)' is inaccessible due to its protection level

Delete the line so your code builds.

3. **Put a breakpoint on the first line of the top-level statements.** Run your program. When it hits the breakpoint, **use Step Into (F10) to execute every line of code one at a time**. Here's what happens:

- It creates an instance of AnimalFeedVendingMachine and calls its Dispense method.
- That method is only defined in the base class, so it calls VendingMachine.Dispense.
- The first line of VendingMachine.Dispense calls the protected CheckAmount method.
- CheckAmount is overridden in the AnimalFeedVendingMachine subclass, which causes VendingMachine.Dispense to call the CheckAmount method defined in AnimalFeedVendingMachine.
- This version of CheckAmount returns true, so Dispense returns the Item property, which has a getter that returns a string. AnimalFeedVendingMachine also overrides this property. It returns the following string: `a handful of animal feed.`

↑

You've been using the debugger to sleuth out bugs in your code. It's also a great tool for learning and exploring C#, like in this "Debug this!" where you can explore how overriding works. Can you think of more ways to experiment with overriding subclasses?

Look, I just don't see why I need to use those "virtual" and "override" keywords. If I don't use them the IDE gives me a warning, but the warning doesn't actually mean anything, because ***my program still runs****! I mean, I'll put the keywords in if it's the "right" thing to do, but it just seems like I'm jumping through hoops* ***for no good reason****.*

There's an important reason for `virtual` and `override`!

The `virtual` and `override` keywords aren't just for decoration. They make a real difference in how your program works. The `virtual` keyword tells C# that a member (like a method, property, or field) can be extended—without it, you can't override it at all. The `override` keyword tells C# that you're extending the member. If you leave out the `override` keyword in a subclass, you're creating a *completely unrelated* method that just *happens to have the same name*.

That sounds a little weird, right? But it actually makes sense—and the best way to really understand how `virtual` and `override` work is by writing code. So let's build a real example to experiment with them.

When a subclass overrides a method in its base class, the more specific version defined in the subclass is always called—even when it's being called by a method in the base class.

Build an app to explore virtual and override

A really important part of inheritance in C# is extending class members. That's how a subclass can inherit some of its behavior from its base class, but override certain members where it needs to—and that's where the **`virtual` and `override` keywords** come in. The `virtual` keyword determines which class members can be extended. When you want to extend a member, you ***must*** use the `override` keyword. Let's create some classes to experiment with `virtual` and `override`. You're going to create a class that represents a safe containing valuable jewels—you'll build a class for some sneaky thieves to steal the jewels.

Do this!

1 **Create a new console app and add the Safe class.**

Here's the code for the Safe class:

```
class Safe
{
    private string contents = "precious jewels";
    private string safeCombination = "12345";

    public string Open(string combination)
    {
        if (combination == safeCombination) return contents;
        return "";
    }

    public void PickLock(Locksmith lockpicker)
    {
        lockpicker.Combination = safeCombination;
    }
}
```

A Safe object keeps valuables in its "contents" field. It doesn't return them unless Open is called with the right combination...or if a locksmith picks the lock.

We're going to add a Locksmith class that can pick the combination lock and get the combination by calling the PickLock method and passing in a reference to itself. The Safe will use its *write-only Combination property* to give the Locksmith the combination.

2 **Add a class for the person who owns the safe.**

The safe owner is forgetful and occasionally forgets their extremely secure safe password. Add a SafeOwner class to represent them:

```
class SafeOwner
{
    private string valuables = "";
    public void ReceiveContents(string safeContents)
    {
        valuables = safeContents;
        Console.WriteLine($"Thank you for returning my {valuables}!");
    }
}
```

3 Add Locksmith class that can pick the lock.

If a safe owner hires a professional locksmith to open their safe, they expect that locksmith to return the contents safe and sound. That's exactly what the Locksmith.OpenSafe method does:

```
class Locksmith
{
    public void OpenSafe(Safe safe, SafeOwner owner)
    {
        safe.PickLock(this);
        string safeContents = safe.Open(Combination);
        ReturnContents(safeContents, owner);
    }

    public string Combination { private get; set; } = "";

    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        owner.ReceiveContents(safeContents);
    }
}
```

The Locksmith's OpenSafe method picks the lock, opens the safe, and then calls ReturnContents to get the valuables safely back to the owner.

4 Add a JewelThief class that wants to steal the valuables.

Uh-oh. Looks like there's a burglar—and the worst kind, one who's also a highly skilled locksmith able to open safes. Add this JewelThief class that extends Locksmith:

```
class JewelThief : Locksmith
{
    private string stolenJewels = "";
    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        stolenJewels = safeContents;
        Console.WriteLine($"I'm stealing the jewels! I stole: {stolenJewels}");
    }
}
```

JewelThief extends Locksmith and inherits the OpenSafe method and Combination property, but its ReturnContents method steals the jewels instead of returning them. INGENIOUS!

5 Add top-level statements that make the JewelThief steal the jewels.

It's time for the big heist! In this code, the JewelThief sneaks into the house and uses its inherited Locksmith.OpenSafe method to get the safe combination. **What do you think will happen when it runs?**

```
SafeOwner owner = new SafeOwner();
Safe safe = new Safe();
JewelThief jewelThief = new JewelThief();
jewelThief.OpenSafe(safe, owner);
Console.ReadKey(true);
```

MINI **Sharpen your pencil**

Read through the code for your program. Before you run it, write down what you think it will print to the console. (Hint: figure out what the JewelThief class inherits from Locksmith.)

A subclass can hide methods in the base class

MINI Sharpen your pencil Solution

Go ahead and run the JewelThief program. Here's what you should see:

```
Thank you for returning my precious jewels!
```

Did you expect the program's output to be different? Maybe something like this:

```
I'm stealing the jewels! I stole: precious jewels
```

C# is supposed to call the most specific method, right? Then why didn't it call JewelThief.ReturnContents?

It looks like the JewelThief object acted just like a Locksmith object! So what happened?

Hiding methods vs. overriding methods

The reason the JewelThief object acted like a Locksmith object when its ReturnContents method was called was because of the way the JewelThief class declared its ReturnContents method. There's a big hint in that warning message you got when you compiled your program:

CS0108	'JewelThief.ReturnContents(string, SafeOwner)' hides inherited member 'Locksmith.ReturnContents(string, SafeOwner)'. Use the new keyword if hiding was intended.

Since the JewelThief class inherits from Locksmith and replaces the ReturnContents method with its own method, it looks like JewelThief is overriding Locksmith's ReturnContents method—but that's not actually what's happening. You probably expected JewelThief to override the method (which we'll talk about in a minute), but instead JewelThief is hiding it.

There's a big difference. When a subclass **hides** a method, it replaces (technically, it redeclares) a method in its base class ***that has the same name***. So now our subclass really has two different methods that share a name: one that it inherits from its base class, and another brand-new one that's defined in that class.

Use the new keyword when you're hiding methods

Take a close look at that warning message. Sure, we know we *should* read our warnings, but sometimes we don't…right? This time, actually read what it says: "Use the new keyword if hiding was intended."

So go back to your program and add the **new** keyword:

```
new protected void ReturnContents(Jewels safeContents, Owner owner)
```

As soon as you add **new** to your JewelThief class's ReturnContents method declaration, that warning message will go away—but your code still won't act the way you expect it to!

It still calls the ReturnContents method defined in the Locksmith class. Why? Because the ReturnContents method is being called **from a method defined by the Locksmith class**—specifically, from inside Locksmith.OpenSafe—even though it's being initiated by a JewelThief object. If JewelThief only hides Locksmith's ReturnContents method, its own ReturnContents method will never be called.

If a subclass just adds a method with the same name as a method in its base class, it only hides the base class method instead of overriding it.

Use different references to call hidden methods

Now we know that JewelThief only ***hides*** the ReturnContents method (as opposed to ***overriding*** it). That causes it to act like a Locksmith object ***whenever it's called like a Locksmith object***. JewelThief inherits one version of ReturnContents from Locksmith, and it defines a second version of it, which means that there are two different methods with the same name. That means your class needs **two different ways to call it**.

There are two different ways to call the ReturnContents method. If you've got an instance of JewelThief, you can use a JewelThief reference variable to call the new ReturnContents method. If you use a Locksmith reference variable to call it, it will call the hidden Locksmith ReturnContents method.

Here's how that works:

```
// The JewelThief subclass hides a method in the Locksmith base class,
// so you can get different behavior from the same object based on the
// reference you use to call it!

// Declaring your JewelThief object as a Locksmith reference causes it to
// call the base class ReturnContents() method.
Locksmith calledAsLocksmith = new JewelThief();
calledAsLocksmith.ReturnContents(safeContents, owner);

// Declaring your JewelThief object as a JewelThief reference causes it to
// call JewelThief's ReturnContents() method instead, because it hides
// the base class's method of the same name.
JewelThief calledAsJewelThief = new JewelThief();
calledAsJewelThief.ReturnContents(safeContents, owner);
```

Can you figure out how to get JewelThief to override the ReturnContents method instead of just hiding it? See if you can do it before reading the next section!

there are no Dumb Questions

Q: I still don't get why they're called "virtual" methods—they seem real to me. What's virtual about them?

A: The name "virtual" has to do with how .NET handles the virtual methods behind the scenes. It uses something called a *virtual method table* (or *vtable*). That's a table that .NET uses to keep track of which methods are inherited and which ones have been overridden. Don't worry—you don't need to know how it works to use virtual methods.

Q: You talked about replacing a superclass with a reference to a subclass. Can you go over that one more time?

A: When you've got a diagram with one class that's above another one, the class that's higher up is **more abstract** than the one that's lower down. More **specific** or **concrete** classes (like Shirt or Car) inherit from more abstract ones (like Clothing or Vehicle). If all you need is a vehicle, a car or van or motorcycle will do. If you need a car, a motorcycle won't be useful to you.

Inheritance works exactly the same way. If you have a method with `Vehicle` as a parameter, and if the Motorcycle class inherits from the Vehicle class, then you can pass an instance of Motorcycle to the method. If the method takes `Motorcycle` as a parameter, you can't pass any Vehicle object, because it may be a Van instance. Then C# won't know what to do when the method tries to access the Handlebars property.

Use the override and virtual keywords to inherit behavior

We really want our JewelThief class to always use its own ReturnContents method, no matter how it's called. This is the way we expect inheritance to work most of the time: a subclass can **override** a method in the base class so the method in the subclass is called instead. Start by using the **`override`** keyword when you declare the ReturnContents method in the JewelThief class:

```
protected override void ReturnContents(string safeContents, SafeOwner owner)
```

But that's not everything you need to do. If you just add the `override` keyword to the class declaration, you'll get a compiler error:

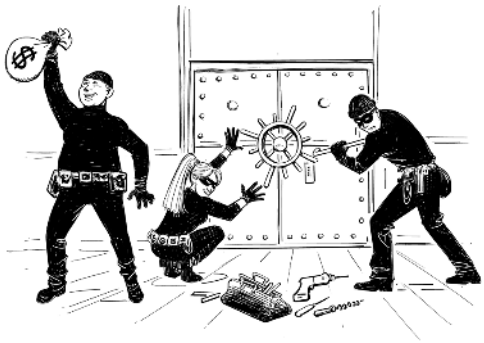
CS0506	'JewelThief.ReturnContents(string, SafeOwner)': cannot override inherited member 'Locksmith.ReturnContents(string, SafeOwner)' because it is not marked virtual, abstract, or override

Take a really close look and read what the error says. JewelThief can't override the inherited member ReturnContents **because it's not marked** `virtual`, `abstract`, or `override` in Locksmith. Well, that's an error we can fix with a quick change. Mark Locksmith's ReturnContents with the **`virtual`** keyword. It should look like this:

```
protected virtual void ReturnContents
    (string safeContents, SafeOwner owner)
```

Now run your program again. Here's the output we've been looking for:

```
I'm stealing the jewels! I stole: precious jewels
```



Sharpen your pencil

Draw a line from each of the following descriptions to the keyword it describes.

A method that can only be **accessed by an instance of the same class**

A method that **a subclass can replace** with a method of the same name

A method that can be **accessed by an instance of any other class**

A method that **hides another method in the superclass** with the same name

A method that **replaces a method in the superclass**

A method that can only be **accessed by a member of the class or its subclass**

`virtual`

`new`

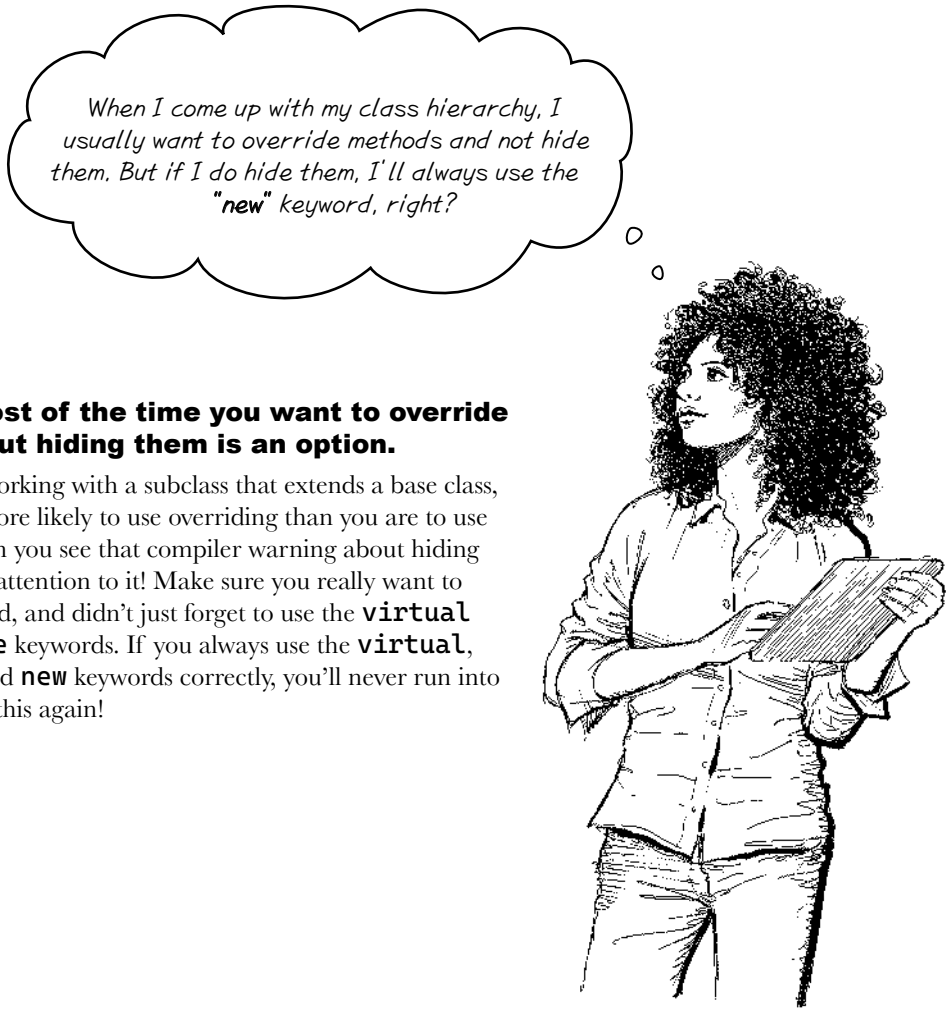
`override`

`protected`

`private`

`public`

1. private 2. virtual 3. public 4. new 5. override 6. protected



Exactly. Most of the time you want to override methods, but hiding them is an option.

When you're working with a subclass that extends a base class, you're much more likely to use overriding than you are to use hiding. So when you see that compiler warning about hiding a method, pay attention to it! Make sure you really want to hide the method, and didn't just forget to use the `virtual` and `override` keywords. If you always use the `virtual`, `override`, and `new` keywords correctly, you'll never run into a problem like this again!

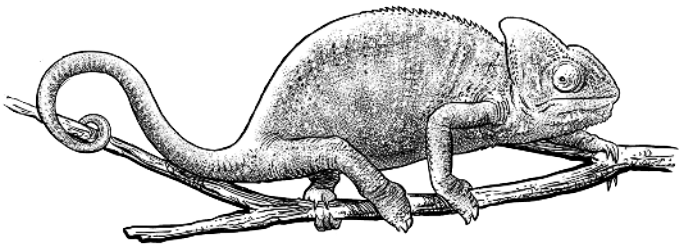
If you want to override a method in a base class, always mark it with the "virtual" keyword, and always use the "override" keyword any time you want to override the method in a subclass. If you don't, you'll end up accidentally hiding methods instead.

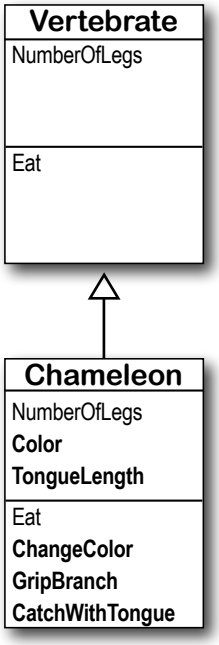
A subclass can access its base class using the base keyword

Even when you override a method or property in your base class, sometimes you'll still want to access it. Luckily, we can use **base**, which lets us access any member of the base class.

1 **All animals eat, so the Vertebrate class has an Eat method that takes a Food object as its parameter.**

```
class Vertebrate {
   public virtual void Eat(Food morsel) {
      Swallow(morsel);
      Digest();
   }
}
```





2 **Chameleons eat by catching food with their tongues. So the Chameleon class inherits from Vertebrate but overrides Eat.**

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      Swallow(morsel);
      Digest();
   }
}
```

This is exactly the same as code in the base class. Do we really need to have two duplicate copies of the same code?

The Chameleon.Eat method needs to call CatchWithTongue, but after that it's identical to the Eat method in the Vertebrate base class that it overrides.

3 **Instead of duplicating the code, we can use the base keyword to call the method that was overridden. Now we have access to both the old and the new version of Eat.**

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      base.Eat(morsel);
   }
}
```

We can't just write "Eat(morsel)" because that would call Chameleon.Eat. We need to use the "base" keyword to access Vertebrate.Eat.

This updated version of the method in the Chameleon class uses the base keyword to *call the Eat method in the base class*. Now we don't have any duplicated code—so if we ever need to change the way all vertebrates eat, chameleons will get the changes automatically.

When a base class has a constructor, your subclass needs to call it

Let's go back to the code you wrote with the Bird, Pigeon, Ostrich, and Egg classes. We want to add a BrokenEgg class that extends Egg, and make 25% of the eggs that the Pigeon lays broken. **Replace the `new` statement *inside* the for loop** in Pigeon.LayEgg with this `if/else` that creates a new instance of either Egg or BrokenEgg:

```
if (Random.Shared.Next(4) == 0)
    eggs[i] = new BrokenEgg(Random.Shared.NextDouble() * 2 + 1, "white");
else
    eggs[i] = new Egg(Random.Shared.NextDouble() * 2 + 1, "white");
```

Add this!

Now we just need a BrokenEgg class that extends Egg. Let's make it identical to the Egg class, except that it has a constructor that writes a message to the console letting us know that an egg is broken:

```
class BrokenEgg : Egg
{
    public BrokenEgg()
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```

Go ahead and **make those two changes** to your Egg program.

Uh-oh—looks like those new lines of code caused compiler errors:

- The first error is on the line where you create a new BrokenEgg: `CS1729 - 'BrokenEgg' does not contain a constructor that takes 2 arguments.`
- The second error is in the BrokenEgg constructor: `CS7036 - There is no argument given that corresponds to the required formal parameter 'size' of 'Egg.Egg(double, string)'.`

This is another great opportunity to ***read those errors*** and figure out what went wrong. The first error is pretty clear: the statement that creates a BrokenEgg instance is trying to pass two arguments to the constructor, but the BrokenEgg class has a parameterless constructor. So go ahead and **add parameters to the constructor**:

```
public BrokenEgg(double size, string color)
```

That takes care of the first error. What about the other error?

Let's break down what that error says:

- It's complaining about `Egg.Egg(double, string)`—this refers to the Egg class constructor.
- It says something about `parameter 'size'`, which the Egg class needs in order to set its Size property.
- But there is `no argument given`, because it's not enough to just modify the BrokenEgg constructor to take arguments that match the parameter. It also needs to **call that base class constructor**.

Modify the BrokenEgg class to **use the `base` keyword to call the base class constructor**:

```
public BrokenEgg(double size, string color) : base(size, color)
```

Now your code compiles. Try running it—now when a Pigeon lays an egg, about a quarter of them will print a message about being broken when they're instantiated (but after that, the rest of the output is the same as before).

A subclass and base class can have different constructors

When we modified BrokenEgg to call the base class constructor, we made its constructor match the one in the Egg base class. What if we want all broken eggs to have a size of zero and a color that starts with the word "broken"? **Modify the statement that instantiates BrokenEgg** so it only takes the color argument:

Modify this!

```
if (Random.Shared.Next(4) == 0)
    eggs[i] = new BrokenEgg("white");
else
    eggs[i] = new Egg(Random.Shared.NextDouble() * 2 + 1, "white");
```

When you make that change, you'll get the "required formal parameter" compiler error again—which makes sense because the BrokenEgg constructor has two parameters, but you're only passing it one argument.

Fix your code **by modifying the BrokenEgg constructor to take one parameter**:

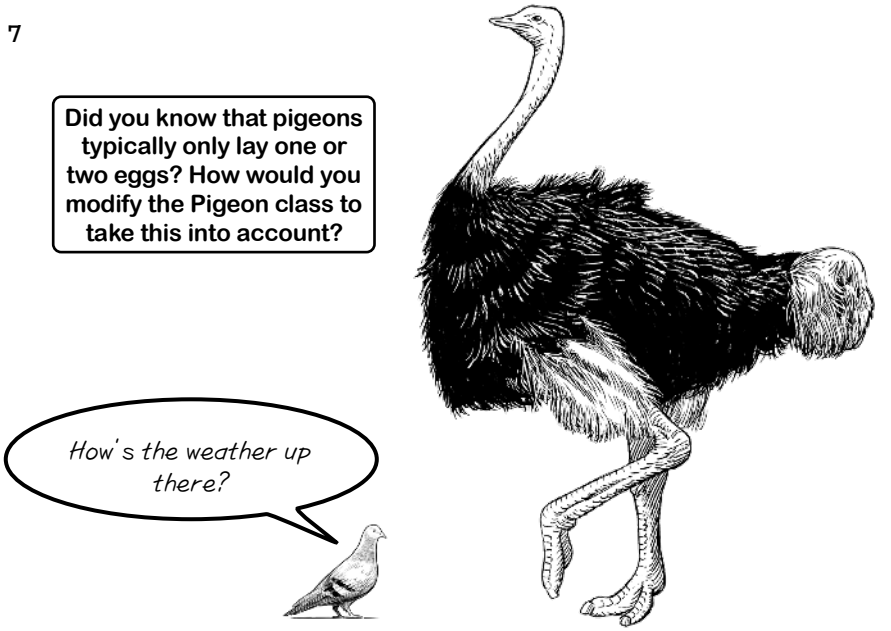
```
class BrokenEgg : Egg
{
    public BrokenEgg(string color) : base(0, $"broken {color}")
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```

The subclass constructor can have any number of parameters, and it can even be parameterless. It just needs to use the `base` keyword to pass the correct number of arguments to the base class constructor.

Now run your program again. The BrokenEgg constructor will still write its message to the console during the `for` loop in the Pigeon constructor, but now it will also cause the Egg to initialize its Size and Color fields. When the `foreach` loop in the top-level statements writes egg.Description to the console, it writes this message for each broken egg:

```
Press P for pigeon, O for ostrich:
p
How many eggs should it lay? 7
A bird laid a broken egg
A bird laid a broken egg
A bird laid a broken egg
A 2.47cm white egg
A 0.00cm broken White egg
A 3.08cm white egg
A 1.41cm white egg
A 0.00cm broken White egg
A 0.00cm broken White egg
A 2.77cm white egg
```

Did you know that pigeons typically only lay one or two eggs? How would you modify the Pigeon class to take this into account?



It's time to finish the job for Owen

The first thing you did in this chapter was modify the damage calculator you built for Owen to roll for damage for either a sword or an arrow. It worked, and your SwordDamage and ArrowDamage classes were well-encapsulated. But aside from a few lines of code, ***those two classes were identical***. You've learned that having code repeated in different classes is inefficient and error-prone, especially if you want to keep extending the program to add more classes for different kinds of weapons. Now you have a new tool to solve this problem: **inheritance**. So it's time for you to finish the damage calculator app. You'll do it in two steps: first you'll design the new class model on paper, and then you'll implement it in code.

Building your class model on paper before you write code helps you understand your problem better so you can solve it more effectively.

Sharpen your pencil

Great code starts in your head, not an IDE. So let's design the class model on paper *before* we start writing code.

We've started you out by filling in the class names. Your job is to add the members to all three classes, and draw the arrows between the boxes.

For reference, we included diagrams for the SwordDamage and ArrowDamage classes that you built earlier. We included the private CalculateDamage method for each class. Make sure to include all public, private, and protected class members when you fill in the class diagram. Write the access modifier (`public`, `private`, or `protected`) next to each class member.

WeaponDamage

The SwordDamage and ArrowDamage classes looked like this at the start of the chapter. They're well-encapsulated, but most of the code in SwordDamage is duplicated in ArrowDamage.

SwordDamage

public Roll
public Magic
public Flaming
public Damage

private
CalculateDamage

ArrowDamage

public Roll
public Magic
public Flaming
public Damage

private
CalculateDamage

SwordDamage

ArrowDamage

We can make some pretty big changes to the way the classes work **without modifying the top-level statements** that call those classes.

When your classes are well-encapsulated, it makes your code *much easier* to modify.

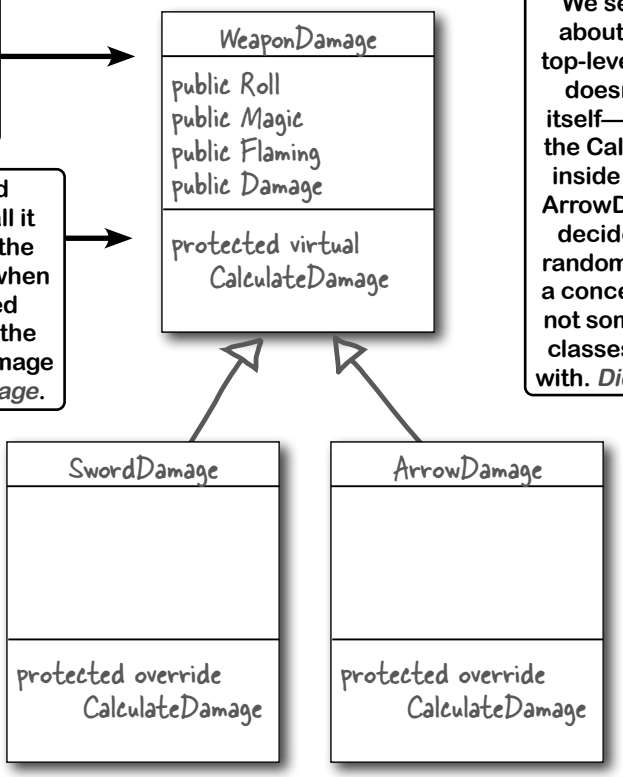
If you know a professional developer, ask them the ***most annoying*** thing they've had to do at work in the last year. There's a pretty good chance that they'll talk about having to make a change to a class, but to do that they had to change these two other classes, which required three other changes, and it was ***hard just to keep track*** of all the changes in their brains. Designing your classes with encapsulation in mind is a great way to avoid ending up in that situation.

Sharpen your pencil Solution

SwordDamage and ArrowDamage had exactly the same properties, so it made sense to move them into the WeaponDamage superclass.

CalculateDamage is marked `virtual`, so the properties call it just like they did before. Now the subclasses will *override* it, so when WeaponDamage.Roll is called from a SwordDamage object, the property calls the CalculateDamage method *defined in SwordDamage*.

We encapsulated the classes earlier in the chapter by marking CalculateDamage `private`. Since the subclasses need to access it, we need to change it to `protected`.



Here's something to think about. We separated the concerns about the user input into the top-level statements. That code doesn't do the calculations itself—they're encapsulated in the CalculateDamage methods inside the SwordDamage and ArrowDamage classes. But we decided that coming up with random numbers for the dice is a concern of the top-level code, not something that the weapon classes need to be concerned with. *Did we make the right call?*

Code can be written in many ways, and usually there's no single "right" answer—not even if it's written in a book! But even if you came up with an equally good solution, let's stick with this class model for the next exercise.

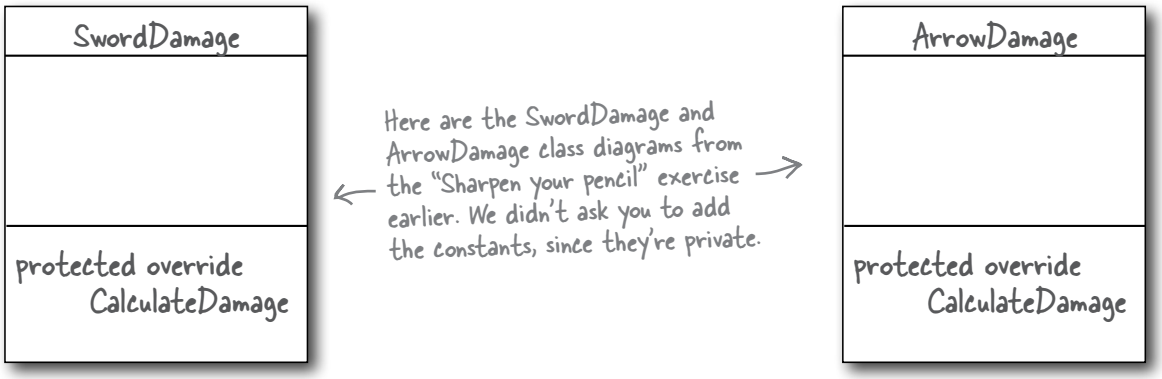
Exercise

Now that you've **designed** the class model, you're ready to write the code to **implement** it. That's a great habit to get into—design your classes first, then turn them into code.

Here's what you'll do to finish the job for Owen. You can reopen the project that you created at the beginning of the chapter, or you can create an entirely new one and copy the relevant parts into it. If your code is very different from the exercise solution earlier in the chapter, you might want to start with the solution code. You can download the code from *https://github.com/head-first-csharp/fifth-edition* if you don't want to type it in.

1. **Don't make any changes to the top-level statements.** It will use the new SwordDamage and ArrowDamage classes exactly like it did at the beginning of the chapter.
2. **Implement the WeaponDamage class.** Add a new WeaponDamage class and make it match the class diagram in the "Sharpen your pencil" solution. Here are a few things to consider:
   - The properties in WeaponDamage are ***almost*** identical to the properties in the SwordDamage and ArrowDamage classes at the beginning of the chapter. There's just a single keyword that's different.
   - Don't put any code in the CalculateDamage class (you can include a comment: `/* the subclass overrides this */`). It needs to be virtual, it can't be private—otherwise you'll get a compiler error:

     CS0621 'WeaponDamage.CalculateDamage()': virtual or abstract members cannot be private
   - Add a constructor that sets the starting roll.
3. **Implement the SwordDamage class.** Here are a few things you need to think about:
   - The constructor has a single parameter, which it passes to the base class constructor.
   - C# always calls the most specific method. That means you'll need to override CalculateDamage and make it do the sword damage calculation.
   - It's worth taking a minute to think about how CalculateDamage works. The Roll, Magic, or Flaming setters call CalculateDamage to make sure the Damage field is updated automatically. Since C# always calls the most specific method, they'll call SwordDamage.CalculateDamage *even though they're part of the WeaponDamage superclass*.
4. **Implement the ArrowDamage class.** It works exactly like SwordDamage, except that its CalculateDamage method does the arrow calculation and not the sword calculation.



Exercise Solution

Here's the code for the WeaponDamage class. The properties are ***almost*** identical to the properties in the old sword and arrow classes. It also has a constructor to set the starting roll, and a CalculateDamage method for the subclasses to override.

WeaponDamage
public Roll public Magic public Flaming public Damage
protected virtual CalculateDamage

```
class WeaponDamage
{
    public int Damage { get; protected set; }

    private int roll;
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    private bool magic;
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    private bool flaming;
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }

    protected virtual void CalculateDamage() { /* the subclass overrides this */ }

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

The Damage property's set accessor needs to be marked `protected`. That way the subclasses have access to update it, but no other class can set it. It's still protected from other classes accidentally setting it, so the subclasses will still be well-encapsulated.

The properties still call the CalculateDamage method, which keeps the Damage property updated. Even though they're defined in the superclass, when they're inherited by a subclass they call the CalculateDamage method defined in that subclass.

This is just like how the JewelThief worked when you had it override the method in LockSmith to steal the jewels from the safe instead of returning them.

The CalculateDamage method itself is empty—we're taking advantage of the fact that C# always calls the most specific method. Now that a SwordDamage class extends WeaponDamage, when its inherited Flaming property's set accessor calls CalculateDamage *it executes the most specific version of that method*, so it calls SwordDamage.CalculateDamage instead.

Exercise Solution

The SwordDamage class extends WeaponDamage and overrides its CalculateDamage method to implement the sword damage calculation. Here's the code:

```
class SwordDamage : WeaponDamage
{
    private const int BASE_DAMAGE = 3;
    private const int FLAME_DAMAGE = 2;

    public SwordDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal magicMultiplier = 1M;
        if (Magic) magicMultiplier = 1.75M;

        Damage = BASE_DAMAGE;
        Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
        if (Flaming) Damage += FLAME_DAMAGE;
    }
}
```

All the constructor needs to do is use the `base` keyword to call the superclass's constructor, using its `startingRoll` parameter as the argument.

And here's the code for the ArrowDamage class. It works just like the SwordDamage class, except it does the calculation for arrows instead:

```
class ArrowDamage : WeaponDamage
{
    private const decimal BASE_MULTIPLIER = 0.35M;
    private const decimal MAGIC_MULTIPLIER = 2.5M;
    private const decimal FLAME_DAMAGE = 1.25M;

    public ArrowDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal baseDamage = Roll * BASE_MULTIPLIER;
        if (Magic) baseDamage *= MAGIC_MULTIPLIER;
        if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
        else Damage = (int)Math.Ceiling(baseDamage);
    }
}
```

Use the debugger to really understand how these classes work

Do this!

One of the most important ideas in this chapter is that when you extend a class, you can override its methods to make pretty significant changes to the way it behaves. Use the debugger to really understand how that works:

- **Set breakpoints** on the lines in the Roll, Magic, and Flaming setters that call CalculateDamage.
- Add a `Console.WriteLine` statement to WeaponDamage.CalculateDamage. *This statement will never get called.*
- Run your program. When it hits any of the breakpoints, use **Step Into** to enter the CalculateDamage method. ***It will step into the subclass***—the WeaponDamage.CalculateDamage method is never called.

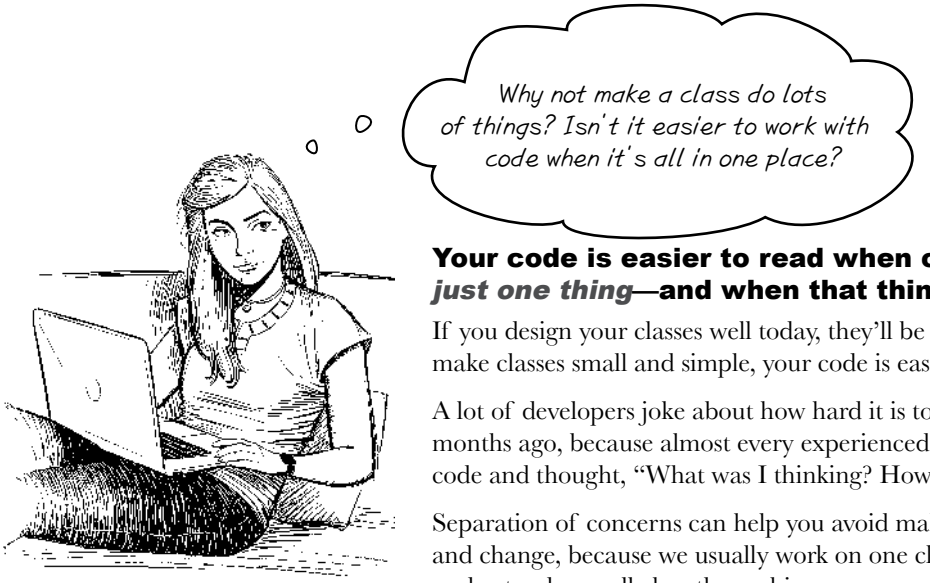
A class should do one thing

When your classes overlap as little as possible, it's called **separation of concerns**. That's an important design principle—it means that each part of your code should focus on doing *just one thing*.

When you're designing classes, separation of concerns is one of the first things you should think about. If one class seems to be doing two different things, try to figure out if you can split it into two classes (or more!).

Imagine if you had a dozen different classes to calculate damage for different weapons. What if you wanted to change Magic from a bool to an int, so you could have weapons with enchantment bonuses (like a +3 magic mace or +1 magic dagger)? With inheritance, you'd just have to change the Magic property in the superclass. Of course, you'd have to modify the CalculateDamage method for each class—but it would be a lot less work, and there's no danger of accidentally forgetting to modify one of the classes. (That happens in professional software development *all the time*!)

This is an example of separation of concerns, because each class has only the code that concerns one specific part of the problem that your program solves. Code that only concerns swords goes into SwordDamage, code only for arrows goes into ArrowDamage, and code that's shared between them goes into WeaponDamage.



Your code is easier to read when classes are small and do *just one thing*—and when that thing is in *just one class*.

If you design your classes well today, they'll be easier to modify later. When you make classes small and simple, your code is easier to write, read, and change.

A lot of developers joke about how hard it is to look at code that you wrote six months ago, because almost every experienced developer has opened up old code and thought, "What was I thinking? How does this even work?"

Separation of concerns can help you avoid making code that's hard to read and change, because we usually work on one class at a time, and it's easier to understand a small class than a big one.

There's another design principle that can help make your code easier to understand and change: **don't repeat yourself**, or **DRY**, which means that any specific thing that your code does should only be in one place. For example, if you have two classes that do the same exact calculation, instead of pasting the same code in both of them, you can put that calculation in a single method and have them both extend the same base class, create an object that has that method, call a static method, or find another way to put it in just one place. That way, if you come back in a few months and need to find the code that does that calculation, you'll know exactly where to find it.

Sens-AI

Use AI to help you research and explore separation of concerns and DRY

We just learned about two important design principles: *separation of concerns* and *don't repeat yourself (DRY)*. These are really valuable ideas that can help you become a better developer. Let's use AI chatbots to explore them.

Do some research about separation of concerns and DRY

AI chatbots can be great research tools. Just don't forget the lesson we learned back in Chapter 2: they aren't perfect, and they make mistakes—so when you're using an AI as a learning and research tool, you really want to be careful to double-check all of the information that it gives you! Use clear, simple prompts, and ask follow-up questions. Some AI chatbots like Copilot also give you links with more information. Click on those links too.

Try these prompts to start your research journey:

- What is "separation of concerns" in programming?
- What is "don't repeat yourself" or "DRY" in programming?
- Give me an example of "separation of concerns" and "DRY" in C#.

Ask the AI who created DRY. These ideas were made by smart people—learn about the folks who came up with them.

Experiment with separation of concerns

In the last exercise, you created three classes: SwordDamage with the behavior that's specific to the sword damage calculation, ArrowDamage with the behavior specific to arrow damage calculation, and WeaponDamage with the behavior that's common to calculating damage for any weapon. What would it look like if you didn't follow the principle of separation of concerns? Ask an AI to show you. Give an AI chatbot this prompt:

```
Combine the following three separate C# classes into a single class.

(paste the code for WeaponDamage class, SwordDamage class, and
ArrowDamage class)
```

Examine the code closely. Do you think it works? **Test it out:** make a backup of your damage calculator app, then delete its WeaponDamage, SwordDamage, and ArrowDamage classes and replace them with the code that the AI generated. Does the app still work? If not, can you fix it? Step through the code—is it easier or harder to understand?

Experiment with DRY

The classes from the last exercise are also a good example of the "don't repeat yourself" principle—but just looking at the code, it's not easy to see exactly why. An AI chatbot can help you with this too. Give the AI this prompt:

```
Combine the following three separate C# classes into two classes that
include the code from the WeaponDamage class instead of extending it.

(paste the code for WeaponDamage class, SwordDamage class, and
ArrowDamage class)
```

Do the same thing you did with the previous code—copy the damage calculator app, delete the damage classes, and replace them with the code that the chatbot generated. Figure out how it works. Can you find the code that it duplicated? What would happen if you had to modify that code—would you remember to change it in both places?

One of the AIs we tested the last prompt with *didn't do what we asked it to*, and generated subclasses that still extended WeaponDamage. We gave it a follow-up prompt: "Rewrite that so SwordDamage and ArrowDamage do not extend WeaponDamage, but instead duplicate its code."

We're about to talk about an important element of game design: dynamics. It's actually such an important concept that it goes beyond game design. In fact, you can find dynamics in almost any kind of app.

Game Design...and Beyond

Dynamics

The **dynamics** of a game describe how the mechanics combine and cooperate to drive the gameplay. Any time you have game mechanics, they lead to dynamics. That's not limited to video games—all games have mechanics, and dynamics arise out of those mechanics.

- We've already seen a **good example of mechanics**: in Owen's role-playing game, he uses formulas (the ones you built into your damage classes) to calculate damage for various weapons. That's a good starting point to think about how a change in those mechanics would affect dynamics.
- What happens if you change the mechanics of the arrow formula so that it multiplies the base damage by 10? That's a small change in mechanics, but it leads to a **huge change in the dynamics** of the game. Suddenly, arrows are much more powerful than swords. Players will stop using swords and start shooting arrows, even at close range—that's a change in dynamics.
- Once the players start **behaving differently**, Owen will need to change his campaigns. For example, some battles designed to be difficult may suddenly be too easy for the players. That makes the players change again.

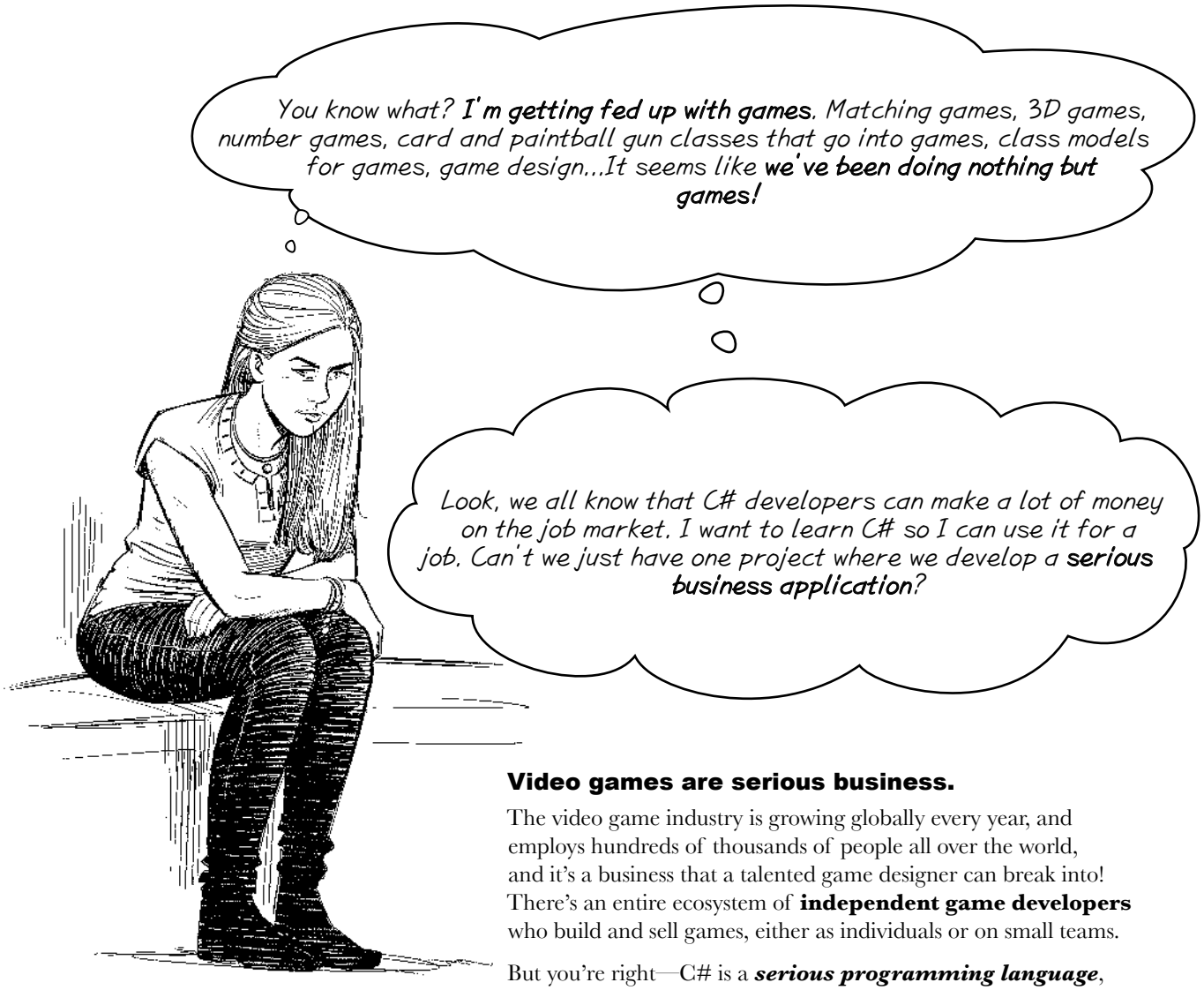
Take a minute to think about all of that. A tiny change to the rules leads to a huge change in the way the players behave. A *small change* in mechanics caused a *very large change in dynamics*. Owen didn't make those changes to gameplay directly; they were follow-on effects from his small rule change. In technical terms, the change in dynamics ***emerged*** from the change in mechanics. We'll explore this in a classic video game.

If you haven't come across the idea of **emergence** before, it may seem a little odd. Let's look at a concrete example:

- The mechanics of Space Invaders—a classic game from 1978—are simple. Aliens march back and forth and fire shots down; if a shot hits the player, they lose a life. The player moves the ship left and right and fires shots up. If a shot hits an alien, it's destroyed. A mothership occasionally flies across the top of the screen for more points. Shields slowly get eaten away by shots. Different aliens add different scores. The aliens march faster as the game goes on. Take a few minutes and watch a gameplay video, or even better play the game.
- The **dynamics of Space Invaders** are more complex. The game starts off very easy—most players can get through the first wave after a few tries—but it quickly gets harder and harder. The only thing that changes is the speed at which the invaders march and shoot. Somehow as the invaders get faster and faster, it changes the entire game. The **tempo**—how fast the game feels—drastically changes.
- Some players try to shoot the aliens starting at the edge of the formation, because the gap at the side of the formation slows down their descent. That's not written anywhere in the code, which just has simple rules for how the invaders march. That's a dynamic, and it's **emergent** because it's a side effect of how the mechanics combine—specifically, the mechanics of how the player shots work combined with the rules for how the invaders march. None of that is programmed into the code of the game. It's not part of the mechanics. It's all dynamics.

Have you ever heard a gamer complain about "balance" in a game—like the choice of weapons in a shooter? Any individual type of gun in a first-person shooter has specific behavior, but add a gun that's too powerful and the entire game changes. That's another example of emergence.

Dynamics can feel like a really abstract concept at first! We'll spend more time on it later in the chapter. For now, keep all of this stuff about dynamics in mind when you're doing the next project. See if you can spot how dynamics come into play as you're coding it.



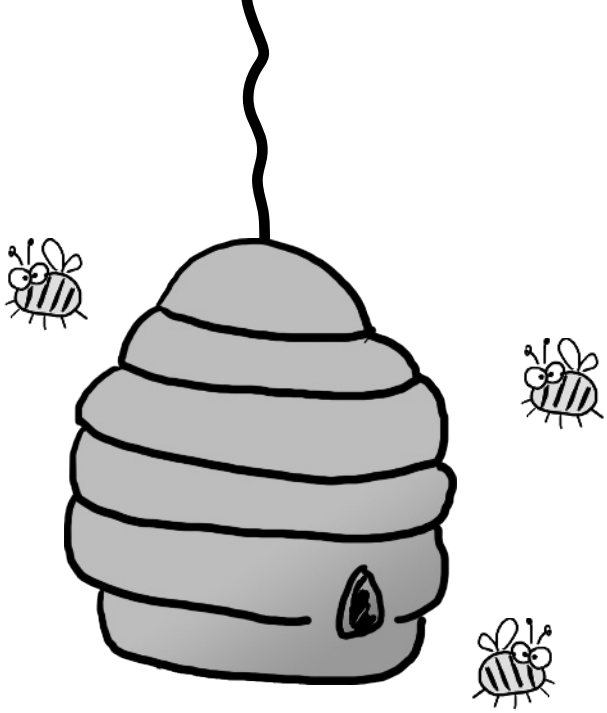
Video games are serious business.

The video game industry is growing globally every year, and employs hundreds of thousands of people all over the world, and it's a business that a talented game designer can break into! There's an entire ecosystem of **independent game developers** who build and sell games, either as individuals or on small teams.

But you're right—C# is a ***serious programming language***, and it's used for all sorts of serious, nongaming applications. In fact, while C# is a favorite language among game developers, it's also one of the most common languages found in businesses across many different industries.

So for this next project, let's get some practice with inheritance by building a ***serious business application***.

Build a beehive management system

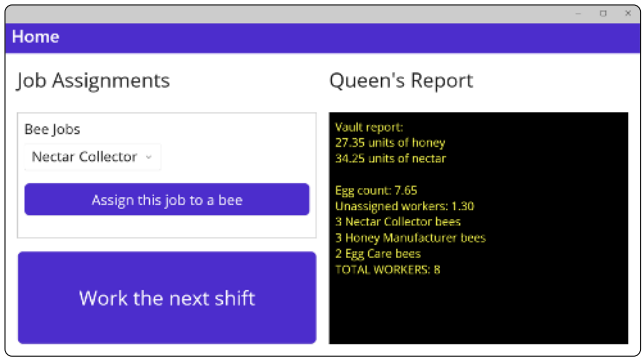


The queen bee needs your help! Her hive is out of control, and she needs a program to help manage her honey production business. She's got a beehive full of workers, and a whole bunch of jobs that need to be done around the hive, but somehow she's lost control of which bee is doing what, and whether or not she's got the beepower to do the jobs that need to be done. It's up to you to build a **beehive management system** to help her keep track of her workers. Here's how it'll work.

1. **The queen assigns jobs to her workers.**
There are three different jobs that the workers can do. **Nectar collector** bees fly out and bring nectar back to the hive. **Honey manufacturer** bees turn that nectar into honey, which bees eat to keep working. Finally, the queen is constantly laying eggs, and **egg care** bees make sure they become workers.

2. **When the jobs are all assigned, it's time to work.**
Once the queen's done assigning the work, she'll tell the bees to work the next shift. At the end of the shift, she gets a shift report that tells her how many bees are assigned to each job and the status of the nectar and honey in the **honey vault**.

3. **Help the queen grow her hive.**
Like all business leaders, the queen is focused on **growth**. The beehive business is hard work, and she measures her hive in the total number of workers. Can you help the queen keep adding workers? How big can she grow the hive before it runs out of honey and she has to file for bee-nkruptcy?



This is a *bigger project* than the ones in the last few chapters.

This is a big project. You can do this!

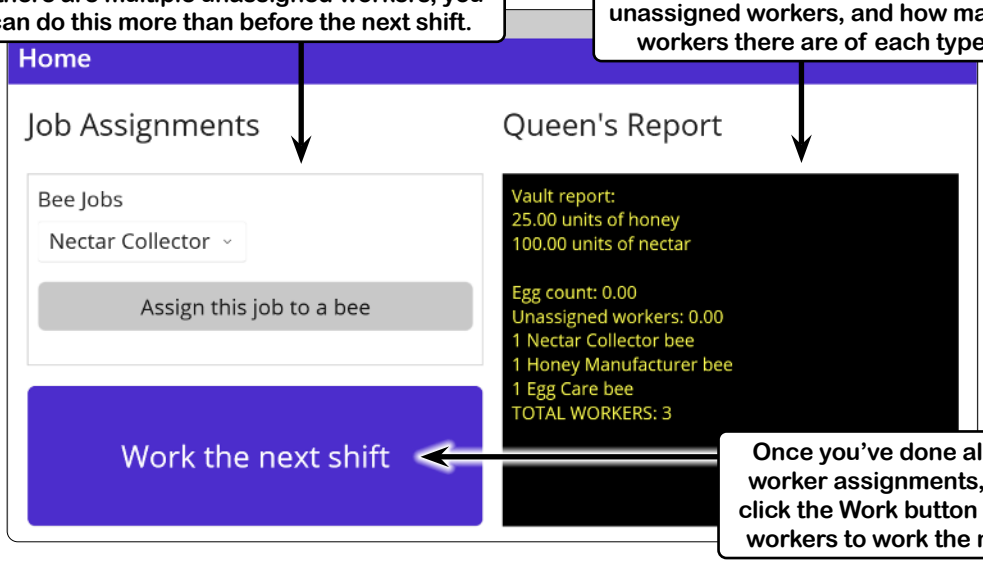
The main goal of this book is to help you learn C#. But we'll also teach important skills that can help you ***become a great developer***. One way to do that is to help show you how to work on—and finish!—larger projects. When you did the Animal Matching Game project in Chapter 1, you broke it down into smaller pieces. You'll do the same for the Beehive Management System project. **First** you'll create the XAML for the main page, **then** you'll do a "Sharpen your pencil" exercise to complete the code for several of the classes, and **finally** you'll do an exercise to finish the rest of the code for the project.

How the Beehive Management System app works

When the app starts, the honey vault has 25 units of honey and 100 units of nectar, and the hive has three workers: a nectar collector bee, a honey manufacturer bee, and an egg care bee. The first shift report delivered is displayed on the righthand side of the app.

If there are any unassigned workers at the start of the shift, you can use the Bee Jobs picker to choose a job, then click the Assign button to give a worker that job (it's disabled if there aren't enough unassigned workers). If there are multiple unassigned workers, you can do this more than before the next shift.

This is the shift report that the queen generates at the end of every shift. It shows the status of the honey vault, followed by the number of eggs, unassigned workers, and how many workers there are of each type.



Once you've done all of your worker assignments, you can click the Work button to tell the workers to work the next shift.

Each worker consumes honey to do a job. The numbers change at the end of each shift to show what they did.

```
Vault report:
27.50 units of honey
34.55 units of nectar

Egg count: 7.80
Unassigned workers: 0.60
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

If the Unassigned Workers count is at least 1, clicking the Assign button assigns a worker to the selected job.

```
Vault report:
30.89 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 1.20
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

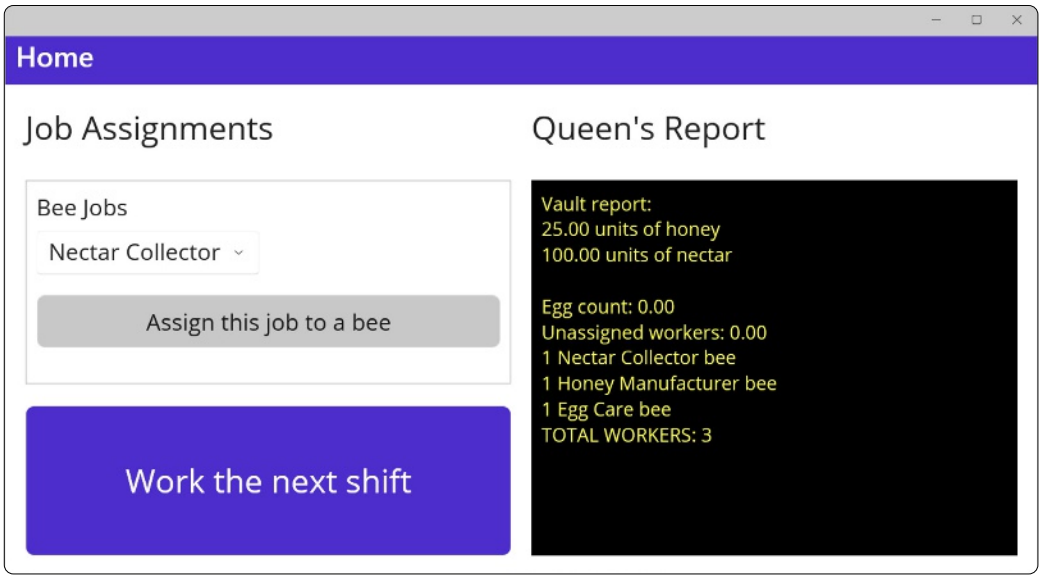
Assigning a worker makes the unassigned workers go down by 1 and the total workers increase by 1.

```
Vault report:
28.74 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 0.20
5 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 10
```

The page uses a grid to lay out the controls for the UI

The Beehive Management System app is a .NET MAUI app. The main page uses a Grid control to lay out the Labels, Buttons, Picker, and other controls.

Take a closer look at it to see how it works:



The Grid has three properties:

- `Margin="20"` adds 20 pixels of space between the controls in the grid and the window frame.
- `ColumnSpacing="20"` adds 20 pixels of space between the two columns.
- `MinimumHeightRequest="400"` keeps the grid from getting smaller than 400 pixels high. If you resize the window smaller than that, the ScrollView will scroll the grid for you.

The Bee Jobs box uses a Border with a VerticalStackLayout

The middle row of the left column has a box that contains a picker and a button. You'll use a Border to draw the box. You need to put two controls inside the Border, but Border controls can only contain one other control, so you'll put a VerticalStackLayout inside it to contain the Picker and Button controls.

```
<Border Padding="10" Margin="0,0,0,20" >
  <VerticalStackLayout>

     <Picker x:Name="JobPicker" ... />
     <Label Text="Bee Jobs"
            FontSize="Medium”/>

     <Button x:Name="AssignJobButton" ... />

  </VerticalStackLayout>
</Border>
```

Exercise

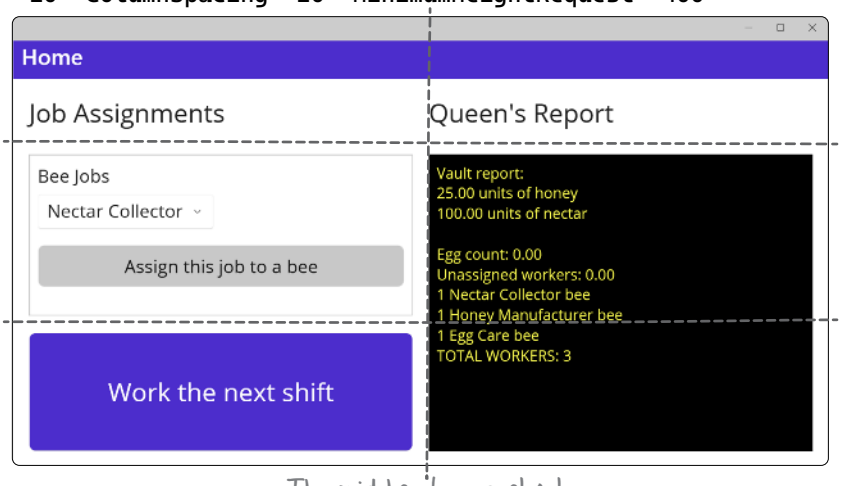
Add the XAML for the main page to your *MainPage.xaml.cs* file. It uses a grid—here's how it's laid out:

```
<ContentPage ... >
    <ScrollView>
        <Grid Margin="20" ColumnSpacing="20" MinimumHeightRequest="400">
```

The grid has three rows.

The middle row has three times the height of the top row.

The bottom row has twice the height of the top row.



The grid has two equal columns.

```
        </Grid>
    </ScrollView>
</ContentPage>
```

The left column of the grid has the following controls:

- The top row has a Label control. Its FontSize property is set to Large.
- The middle row has a Border control. It contains a VerticalStackLayout with a Label with FontSize set to Medium, a Picker, and a Button. Use x:Name to name the Picker JobPicker and name the button AssignJobButton.
- The Picker has a 20 pixel bottom margin. They both have FontSize set to Medium. Look at the screenshot to figure out the Picker's Title and the Button's Text properties. Add a SemanticProperties.Hint to each of them.
- Modify the MainPage constructor in *MainPage.xaml.cs* to set JobPicker.ItemsSource to a new array that contains the following strings: `"Nectar Collector", "Honey Manufacturer", "Egg Care"`
- The bottom row has a Button control. Use x:Name to name it WorkShiftButton. Its font size is Large.
- The bottom row has a ***second* Button control**. You can't see it because it has the `IsVisible="False"` property. Set its BackgroundColor property to Red, its FontSize to Large, and use x:Name to name it OutOfHoneyButton. It should display the following text: `"The hive is out of honey"`
- Add an event handler method for all three of the Button controls. Make sure they all have semantic hints.

The right column of the grid has the following controls:

- The top row has a Label control. Its FontSize property is set to Large.
- The bottom two rows have another Label control with RowSpan="2" so it spans both rows. Its background color is set to Black, and text color is set to Yellow. It has the Padding="10" property to give space between the edge and the text, and VerticalOptions="FillAndExpand" so it fills the cell. Use x:Name to name it StatusReport.

Exercise Solution

Here's the XAML for the main page. Did yours come out a little different than ours? That's OK! There are a lot of different ways to design this page—if it still works and you like the way it looks, then you got it right.

```
<ScrollView>
  <Grid Margin="20" ColumnSpacing="20" MinimumHeightRequest="400">
    <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="3*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Label Text="Job Assignments" FontSize="Large" />

    <Border Grid.Row="1" Padding="10" Margin="0,0,0,20" >
        <VerticalStackLayout>

            <Label Text="Bee Jobs" FontSize="Medium"/>
            <Picker x:Name="JobPicker"
                 FontSize="Medium" Margin="0,0,0,20"
                SemanticProperties.Hint="Lets the user pick a job to assign" />

            <Button x:Name="AssignJobButton" Text="Assign this job to a bee"
                FontSize="Medium" Clicked="AssignJobButton_Clicked"
                SemanticProperties.Hint="Assigns the job to a worker bee" />

        </VerticalStackLayout>
    </Border>

    <Button x:Name="WorkShiftButton" Grid.Row="2" Text="Work the next shift"
            FontSize="Large" Clicked="WorkShiftButton_Clicked"
            SemanticProperties.Hint="Makes the bees work the next shift" />

    <Button x:Name="OutOfHoneyButton" Grid.Row="2" Text="The hive is out of honey"
            BackgroundColor="Red" FontSize="Large" IsVisible="False"
            Clicked="OutOfHoneyButton_Clicked"
            SemanticProperties.Hint="Restarts the Beehive Management System" />

    <Label Text="Queen's Report" Grid.Column="1" FontSize="Large"  />

    <Label x:Name="StatusReport" Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"
           VerticalOptions="FillAndExpand" FontSize="Small"
           BackgroundColor="Black" TextColor="Yellow" Padding="10" />
  </Grid>
</ScrollView>
```

The middle row is three times as tall as the top row, and the bottom row is twice as tall.

The combination of the Border and VerticalStackLayout makes both the Picker and Button appear inside a box.

The "Work the next shift" button will be visible while the hive is working. If the hive runs out of honey, the app will make it invisible and show the other button to let the user restart the hive.

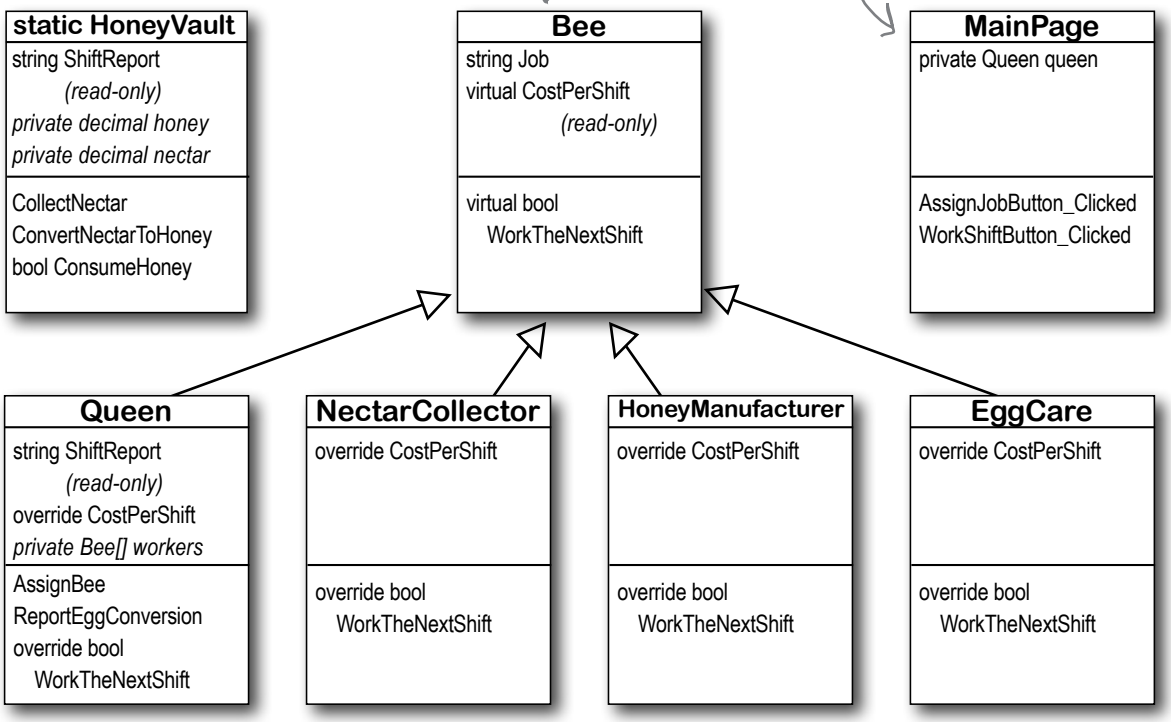
The Beehive Management System class model

Here are the classes that you'll build for the Beehive Nanagement System. There's an inheritance model with a base class and four subclasses, a static class to manage the honey and nectar that drive the hive business, and the MainPage class with the code-behind for the main page.

HoneyVault is a static class that keeps track of the honey and nectar in the hive. Bees use the ConsumeHoney method, which checks if there's enough honey to do their jobs, and if so subtracts the amount requested.

Bee is the base class for all of the bee classes. Its WorkTheNextShift method calls the Honey Vault's ConsumeHoney method to make sure each bee consumes the honey they need to do their jobs. If there's not enough honey, it returns false.

The code-behind for the main page just does a few things. It creates an instance of Queen, and has Click event handlers for the buttons to call her WorkTheNextShift and AssignBee methods and display the shift report.



This Bee subclass uses an array to keep track of the workers and calls each of their WorkTheNextShift methods to make them do their jobs. Queen also has private methods, which we'll talk about later.

This Bee subclass overrides WorkTheNextShift to call the HoneyVault method to collect nectar.

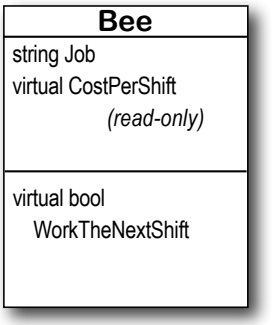
This Bee subclass overrides WorkTheNextShift to call the HoneyVault method to convert nectar to honey.

This Bee subclass keeps a reference to the Queen object, and overrides WorkTheNextShift to call the Queen's ReportEggConversion method.

You'll use decimal fields to keep track of honey and nectar, for the same reasons you'd use decimal for money.

All bees in the system extend the Bee class

This is the Bee class. It's the **superclass of all the other bee classes in the system**. Each shift, every bee—the queen and every one of her workers—consumes honey and does their job. The system does this by calling the WorkTheNextShift method, which consumes the honey from the vault using the CostPerShift property. The WorkTheNextShift method is marked **`virtual`** because each subclass will extend it to do its specific job.




```
class Bee
{
    /// <summary>
    /// The units of honey this bee consumes per shift
    /// </summary>
    public virtual decimal CostPerShift { get; }

    /// <summary>
    /// The job assigned to this bee
    /// </summary>
    public string Job { get; private set; }

    /// <summary>
    /// The Bee class constructor
    /// </summary>
    /// <param name="job">The job to assign to this bee</param>
    public Bee(string job)
    {
        Job = job;
    }

    /// <summary>
    /// If there's enough honey for this bee to work the next shift, consume the honey
    /// </summary>
    /// <returns>True if there was enough honey to do the job, false otherwise</returns>
    public virtual bool WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
            return true;
        else
            return false;
    }
}
```

Each subclass will override CostPerShift to return a different value. Egg care work is just as important as collecting honey, but flying around and collecting nectar takes more energy than caring for eggs, so the NectarCollector's CostPerShift property will return a higher value than the EggCare's CostPerShift property.

Every bee needs to consume honey, so the WorkTheNextShift method consumes the amount of honey that particular bee needs. Every bee class does a different job, so they all consume different amounts of honey—and since C# always calls the *most specific member*, WorkTheNextShift will use the CostPerShift property to figure out how much honey to consume, and each subclass's CostPerShift property will return the amount of honey consumed for that specific type of bee.

All the constants are in their own static class

Each of the bees consumes honey and does a job. These constants determine how both of those things work.

```
static class Constants
{
    /// <summary>
    /// The amount of honey the queen consumes each shift
    /// </summary>
    public const decimal QUEEN_COST_PER_SHIFT = 2.15m;

    /// <summary>
    /// Number of eggs the queen lays per shift
    /// </summary>
    public const decimal EGGS_PER_SHIFT = 0.45m;

    /// <summary>
    /// The shift report shows a warning if the honey level drops below this
    /// </summary>
    public const decimal LOW_LEVEL_WARNING = 10m;

    /// <summary>
    /// How much honey each unassigned worker consumes
    /// </summary>
    public const decimal HONEY_PER_UNASSIGNED_WORKER = 0.5m;

    /// <summary>
    /// Cost of a nectar collector per shift
    /// </summary>
    public const decimal NECTAR_COLLECTOR_COST = 1.95m;

    /// <summary>
    /// How much nectar a nectar collector can collect per shift
    /// </summary>
    public const decimal NECTAR_COLLECTED_PER_SHIFT = 33.25m;

    /// <summary>
    /// Cost of a honey manufacturer per shift
    /// </summary>
    public const decimal HONEY_MANUFACTURER_COST = 1.7m;

    /// <summary>
    /// How much nectar the honey manufacturer processes per shift
    /// </summary>
    public const decimal NECTAR_PROCESSED_PER_SHIFT = 33.15m;

    /// <summary>
    /// How many units of honey get converted from one unit of nectar
    /// </summary>
    public const decimal NECTAR_CONVERSION_RATIO = .19m;

   /// <summary>
    /// Cost of an egg care worker per shift
    /// </summary>
    public const decimal EGG_CARE_COST = 1.35m;

    /// <summary>
    /// Number of eggs an egg care worker converts to unassigned workers per shift
    /// </summary>
    public const decimal CARE_PROGRESS_PER_SHIFT = 0.15m;
}
```

Every shift the queen consumes a certain amount of honey. During the shift she lays a certain number of eggs. These constants determine exactly how much honey she consumes and how many eggs she lays each shift.

Every unassigned worker consumes honey per shift, but less than the rest of the workers because they're sitting around waiting to do a job.

The nectar collector class will use these constants to determine how much each worker costs and how much nectar they collect during the shift.

These constants determine how much honey a honey manufacturer processes each shift, and how much they consume.

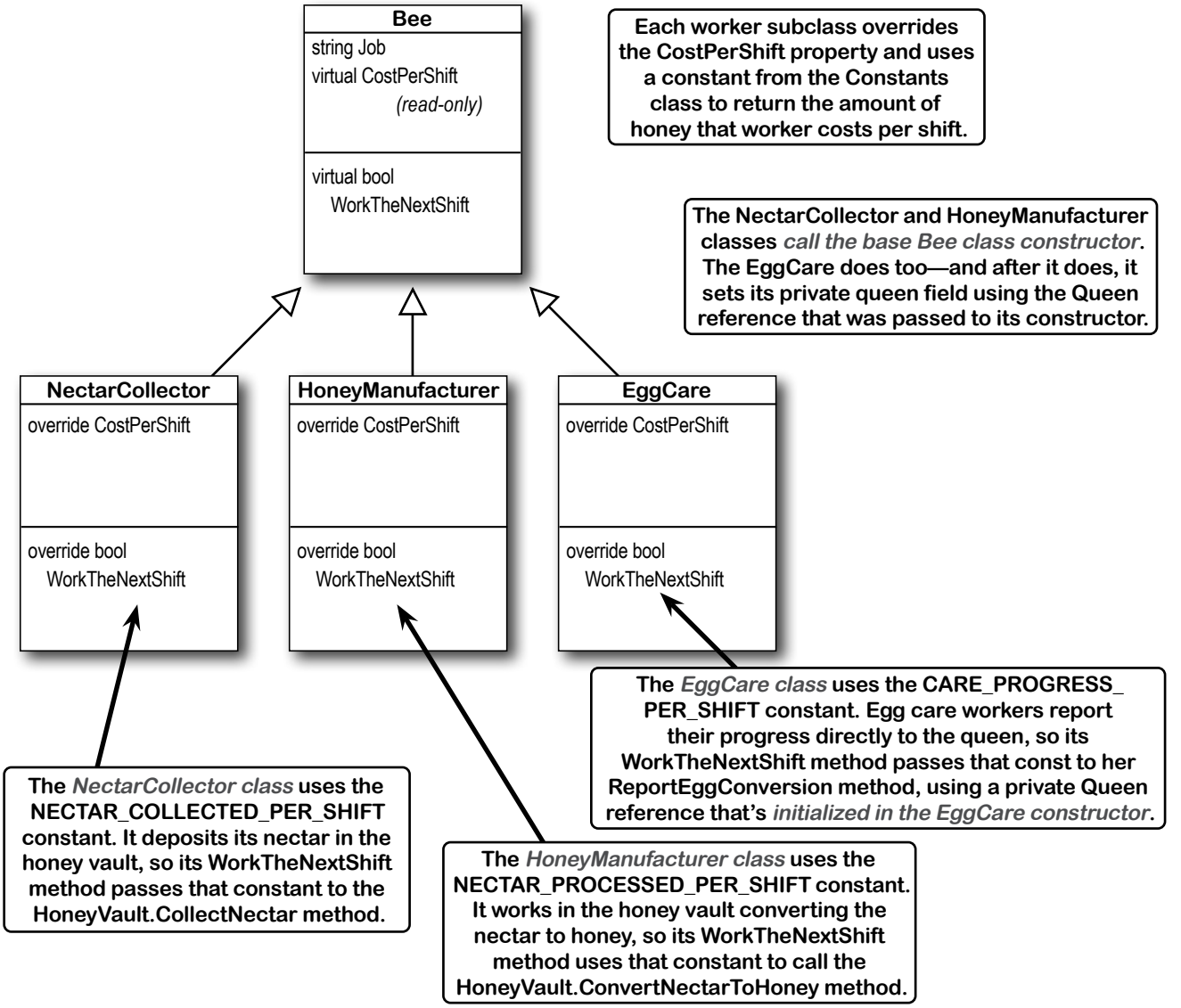
This is used by the honey vault in its method to convert nectar to honey.

This determines how many shifts it takes an egg care worker to bring an egg to maturity.

The worker bees extend the Bee class

Workers can be assigned **one of three jobs**: nectar collectors add nectar to the honey vault, honey manufacturers convert the nectar into honey, and egg care bees turn eggs into workers who can be assigned to jobs. During each shift, the queen lays eggs (just under two shifts per egg). The queen updates her shift report at the end of the shift. It shows the honey vault status and the number of eggs, unassigned workers, and bees assigned to each job.

Here's a closer look at how the NectarCollector, HoneyManufacturer, and EggCare classes work. They're all subclasses of the Bee class. They all use constants defined in the static Constants class. Each bee does a different job, so they all have different WorkTheNextShift methods that override Bee.WorkTheNextShift to do their jobs.



Sharpen your pencil

Fill in the missing parts of the Bee subclasses based on what we've told you about how they work.

```
class NectarCollector : Bee
{
    public NectarCollector() : ______________________

    public override decimal CostPerShift {
        get { return ________________________________ }
    }

    public override bool WorkTheNextShift()
    {
        ____________________(____________________________________);
        return base.WorkTheNextShift();
    }
}

class HoneyManufacturer : Bee
{
    public HoneyManufacturer() : _________________________

    public override decimal CostPerShift {
      get { return ________________________________ }
    }

    public override bool WorkTheNextShift()
    {
        __________________________(____________________________________);
        return base.WorkTheNextShift();
    }
}

class EggCare : Bee
{
    private Queen queen;

    public EggCare(Queen queen) : ________________
    {
        this.queen = queen;
    }

    public override decimal CostPerShift {
        get { return ________________________; }
    }

    public override bool WorkTheNextShift()
    {
        ____________________(________________________________);
        return base.WorkTheNextShift();
    }
}
```

Each Bee subclass overrides WorkTheNextShift so it first does its job, then consumes its honey—so this statement is what makes the NectarCollector bee do its job.

The only difference between these bees is the job they do and how much honey they consume, so it makes sense that they all extend the Bee class and override the CostPerShift property and the WorkTheNextShift method.

Sharpen your pencil Solution

Fill in the missing parts of the Bee subclasses based on what we've told you about how they work.

```
class NectarCollector : Bee
{
    public NectarCollector() : base("Nectar Collector") { }

    public override decimal CostPerShift {
        get { return Constants.NECTAR_COLLECTOR_COST; }
    }

    public override bool WorkTheNextShift()
    {
        HoneyVault.CollectNectar(Constants.NECTAR_COLLECTED_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}

class HoneyManufacturer : Bee
{
    public HoneyManufacturer() : base("Honey Manufacturer") { }

    public override decimal CostPerShift {
      get { return Constants.HONEY_MANUFACTURER_COST; }
    }

    public override bool WorkTheNextShift()
    {
        HoneyVault.ConvertNectarToHoney(Constants.NECTAR_PROCESSED_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}

class EggCare : Bee
{
    private Queen queen;

    public EggCare(Queen queen) : base("Egg Care")
    {
        this.queen = queen;
    }

    public override decimal CostPerShift {
        get { return Constants.EGG_CARE_COST; }
    }

    public override bool WorkTheNextShift()
    {
        queen.ReportEggConversion(Constants.CARE_PROGRESS_PER_SHIFT);
        return base.WorkTheNextShift();
    }
}
```

All three classes call the base constructor, passing it a string that's used to figure out which job the bee is doing.

The NectarCollector's WorkTheNextShift method passes the NECTAR_COLLECTED_PER_SHIFT constant to the HoneyVault.CollectNectar method.

The HoneyManufacturer's WorkTheNextShift method passes the NECTAR_PROCESSED_PER_SHIFT constant to the HoneyVault.ConvertNectarToHoney method.

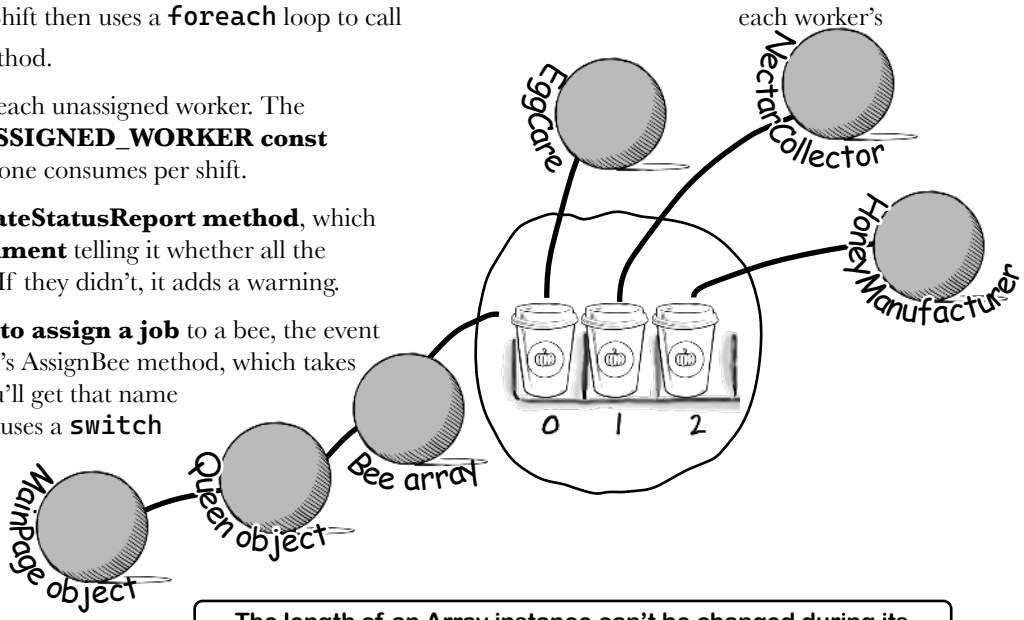
All three classes override the CostPerShift property and have it return a constant from the static Constants class.

Each EggCare object calls the queen's ReportEggConversion method to tell the queen how many eggs they converted into unassigned worker bees.

The Queen class: how she manages the worker bees

When you **press the button to work the next shift**, the button's event handler calls the Queen object's WorkTheNextShift method, which is inherited from the Bee base class. Here's what happens next:

- **Bee.WorkTheNextShift** calls HoneyVault.ConsumeHoney(HoneyConsumed), using the CostPerShift property (which each subclass overrides with a different value) to determine how much honey she needs.
- The queen **overrides WorkTheNextShift** so that each shift she adds 0.45 eggs to her private eggs field (using the EGGS_PER_SHIFT constant). The egg care bee will call her ReportEggConversion method to tell the queen to update the egg and worker counts. She'll then decrease eggs and increase unassignedWorkers.
- Queen.WorkTheNextShift then uses a `foreach` loop to call WorkTheNextShift method.
- It consumes honey for each unassigned worker. The **HONEY_PER_UNASSIGNED_WORKER const** tracks how much each one consumes per shift.
- Finally, it calls its **UpdateStatusReport method**, which takes **a Boolean argument** telling it whether all the workers did their jobs. If they didn't, it adds a warning.



When you **press the button to assign a job** to a bee, the event handler calls the Queen object's AssignBee method, which takes a string with the job name (you'll get that name from the JobPicker control). It uses a `switch` statement to create a new instance of the appropriate Bee subclass and pass it to AddWorker, so make sure you **add the AddWorker method** that follows to your Queen class.

The length of an Array instance can't be changed during its lifetime. That's why C# has this useful static *Array.Resize method*. It doesn't actually resize the array. Instead, it creates a new one and copies the contents of the old one into it. Notice how it uses the `ref` keyword—we'll learn more about that later in the book.

You'll need this AddWorker method to add a new worker to the queen's worker array. It calls Array.Resize to expand the array, then adds the new worker bee to it.

```
/// <summary>
/// Expand the workers array by one slot and add a Bee reference.
/// </summary>
/// <param name="worker">Worker to add to the workers array.</param>
private void AddWorker(Bee worker)
{
    if (unassignedWorkers >= 1)
    {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
```

Here's the code-behind for MainPage.xaml.cs

This code will help you figure out how to write the code for the HoneyVault and Queen classes.

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        UpdateStatusAndEnableAssignButton();
    }

    private void UpdateStatusAndEnableAssignButton()
    {
        StatusReport.Text = queen.StatusReport;
        AssignJobButton.IsEnabled = queen.CanAssignWorkers;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
        {
            WorkShiftButton.IsVisible = false;
            OutOfHoneyButton.IsVisible = true;
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
        }

        UpdateStatusAndEnableAssignButton();
        SemanticScreenReader.Default.Announce(queen.StatusReport);
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
        UpdateStatusAndEnableAssignButton();
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        WorkShiftButton.IsVisible = true;
        OutOfHoneyButton.IsVisible = false;
        UpdateStatusAndEnableAssignButton();
    }
}
```

The UpdateStatusAndEnableAssignButton method does two things. It uses the Queen object's StatusReport property to set the text of the label that displays the status report. Then it uses the CanAssignWorkers property to enable or disable the button that assigns the job—which means that property needs to return True if there's at least one unassigned worker, or False if there isn't.

If the honey vault runs out of honey, the Queen object's WorkTheNextShift method returns false. If that happens, the bees can't work the next shift, so it hides the button to work the next shift and shows the button to reset the Queen and the Honey Vault.

The "Work the next shift" button announces the status so people using screen readers get the information they need to manage the hive.

The "assign job" button passes the text from the selected Picker item directly to Queen.AssignBee, so it's really important that the cases in the switch statement match the Picker items exactly.

This button's event handler resets the Queen and the Honey Vault, and sets the controls back to the way they were when the app started.

Exercise

This is the biggest exercise we've given you so far. You can do this! Just take it one step at a time.

Add the Constants class we gave you earlier, then add a static class called HoneyVault to manage the honey

- Look back at the "Sharpen your pencil" solution to see how the Bee subclasses call methods in HoneyVault.
- **Add two constants** to the static Constants class (`INITIAL_HONEY = 25f` and `INITIAL_NECTAR = 100f`) and use them to initialize the two private fields called `honey` and `nectar`.
- The **ConsumeHoney method** is how the bees use honey to do their jobs. It takes a parameter, `amount`. If honey is greater than or equal to `amount`, it subtracts amount from honey and returns true; otherwise it returns false.
- The **CollectNectar method** is called by the NectarCollector bee each shift. It takes a parameter, `amount`. If `amount` is greater than zero, it adds it to the nectar field.
- The **ConvertNectarToHoney method** converts nectar to honey. It takes a decimal parameter called `amount`, subtracts that amount from its nectar field, and adds amount × NECTAR_CONVERSION_RATIO to the honey field. (If the amount passed to the method is greater than the nectar left in the vault, it converts the remaining nectar.)
- The **StatusReport property** only has a get accessor that returns a string with separate lines with the amount of honey and the amount of nectar in the vault. If the honey is below LOW_LEVEL_WARNING, it adds a warning (`"LOW HONEY – ADD A HONEY MANUFACTURER"`). It does the same for the nectar field.

Add the HoneyManufacturer, NectarCollector, and BeeCare classes, then create the Queen class

- The "Sharpen your pencil" solution has the code for the first three Bee subclasses. The Queen class is the other subclass of Bee. We showed you how the Queen class works (and gave you an AddWorker method that adds workers to an array of Bee objects). Also look carefully at the code-behind to see how it's used.
- The Queen class has a **private Bee[] field** called workers. It starts off as an empty array. We gave you the AddWorker method to add to it—you'll create the AssignBee method that instantiates Bee objects and calls it.
- There are two **private decimal fields** called eggs and unassignedWorkers to keep track of the number of eggs (which she adds to each shift) and the number of workers waiting to be assigned.
- The queen starts off with three unassigned workers (so set unassignedWorkers to 3) and zero eggs. The Queen's **constructor** calls the AssignBee method three times to create three worker bees, one of each type.
- The **AssignBee method** takes a `string?` parameter with a job name (like `"Egg Care"`). It has `switch (job)` with cases that call the AddWorker method that we gave you when we described the Queen class. For example, if `job` is `"Egg Care"` then it calls AddWorker(new EggCare(this)).
- Go back to the "Sharpen your pencil" solution and look at how the egg care bees call the queen's **ReportEggConversion method**. The method takes a decimal parameter called `eggsToConvert`. It checks if the eggs field is >= eggsToConvert—if it is, it subtracts eggsToConvert from eggs and adds it to unassignedWorkers.
- She overrides the **WorkTheNextShift method** to add eggs, tell the worker bees to work, and feed honey to the unassigned workers waiting for work. The EGGS_PER_SHIFT constant (set to 0.45f) is added to the eggs field. She uses a `foreach` loop to call each worker's WorkTheNextShift method. Then she calls HoneyVault.ConsumeHoney, passing it the constant HONEY_PER_UNASSIGNED_WORKER (set to 0.5f) × unassignedWorkers. The loop keeps track of whether all of the workers did their jobs, and passes that to UpdateStatusReport.
- Look carefully at the shift reports in the screenshots. The private **UpdateStatusReport method** generates it (using HoneyVault.StatusReport). She calls UpdateShiftReport at the end of her WorkTheNextShift and AssignBee methods. UpdateShiftReport calls a **private WorkerStatus method** with a string parameter called `job` ("Nectar Collector") and returns a string with the number of workers doing that job ("3 Nectar Collector bees"). It also takes a bool parameter—if it's true, it adds `"WARNING: NOT ALL WORKERS DID THEIR JOBS"` to the end of the report.

If you get stuck, it's not cheating to look at our solution. That's a great way to learn too!

Exercise Solution

Here are the constants that were added to the static Constants class. They're used to initialize and reset HoneyVault.

```
static class Constants
{
    /// <summary>
    /// The amount of honey the hive starts with
    /// </summary>
    public const decimal INITIAL_HONEY = 25m;

    /// <summary>
    /// The amount of nectar the hive starts with
    /// </summary>
    public const decimal INITIAL_NECTAR = 100m;
```

Here's the Queen class, a Bee subclass that assigns workers, tells them to do their jobs, and makes status reports:

```
class Queen : Bee
{
    private Bee[] workers = new Bee[0];
    private decimal eggs = 0;
    private decimal unassignedWorkers = 3;

    public bool CanAssignWorkers { get { return unassignedWorkers >= 1; } }

    public string StatusReport { get; private set; } = "";

    public override decimal CostPerShift {
        get { return Constants.QUEEN_COST_PER_SHIFT; } }

    public Queen() : base("Queen")
    {
        AssignBee("Nectar Collector");
        AssignBee("Honey Manufacturer");
        AssignBee("Egg Care");
    }

    public void AssignBee(string? job)
    {
        switch (job)
        {
            case "Nectar Collector":
                AddWorker(new NectarCollector());
                break;
            case "Honey Manufacturer":
                AddWorker(new HoneyManufacturer());
                break;
            case "Egg Care":
                AddWorker(new EggCare(this));
                break;
        }
        UpdateStatusReport(true);
    }
```

The queen's private workers is a Bee array that starts empty. It will get initialized with Bee objects in the constructor.

If you don't give StatusReport a default value, you'll get that warning about it being non-null when exiting the constructor.

The queen starts things off by assigning one bee of each type in her constructor.

The AssignBee method takes a job string as a parameter. It uses that string with a switch statement to figure out what kind of worker to create, then it instantiates the right Bee subclass and calls the AddWorker method that we gave you to add it to the workers array.

Take a few minutes and use the debugger to really understand how this works. Add a *breakpoint to each Bee subclass's WorkTheNextShift method* and step through them.

Exercise Solution

We gave you this AddWorker method. It resizes the array and adds a Bee to the end.

```
    private void AddWorker(Bee worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private void UpdateStatusReport(bool allWorkersDidTheirJobs)
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"\nEgg count: {eggs:0.00}\nUnassigned workers: {unassignedWorkers:0.00}\n" +
        $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
        $"\n{WorkerStatus("Egg Care")}\nTOTAL WORKERS: {workers.Length}";

        if (!allWorkersDidTheirJobs)
            StatusReport += "\nWARNING: NOT ALL WORKERS DID THEIR JOBS";
    }

    private string WorkerStatus(string job)
    {
        int count = 0;
        foreach (Bee worker in workers)
            if (worker.Job == job) count++;
        string s = "s";
        if (count == 1) s = "";
        return $"{count} {job} bee{s}";
    }

   public void ReportEggConversion(decimal eggsToConvert)
    {
        if (eggs >= eggsToConvert)
        {
            eggs -= eggsToConvert;
            unassignedWorkers += eggsToConvert;
        }
    }

    public override bool WorkTheNextShift()
    {
        eggs += Constants.EGGS_PER_SHIFT;
        bool allWorkersDidTheirJobs = true;
        foreach (Bee worker in workers)
        {
            if (!worker.WorkTheNextShift())
                allWorkersDidTheirJobs = false;
        }
        HoneyVault.ConsumeHoney(unassignedWorkers * Constants.HONEY_PER_UNASSIGNED_WORKER);
        UpdateStatusReport(allWorkersDidTheirJobs);
        return base.WorkTheNextShift();
    }
}
```

Try doing a global search and replace to change "decimal" to "float," then add a breakpoint to the first line of AddWorker. You'll see unassignedWorkers is sometimes equal to 0.999999...—this is just like the 0.30000000000000004 problem we talked about in Chapter 4. That's why we asked you to use decimal for number values.

You had to look really closely at the shift report in the screenshot to figure out what to include here.

The instructions to add the UpdateStatusReport method didn't tell you to create this WorkerStatus method. We added it because we felt it made the code easier to understand. Your UpdateStatusReport might look really different from ours—and that's OK!

The "Sharpen" solution showed you how the egg care bees call the ReportEggConversion method to tell the queen how many eggs have been converted into unassigned workers.

The constants used in the Queen class are really important because they determine how the program behaves over the course of many shifts. If she lays too many eggs, they eat more honey, but also speed up progress. If unassigned workers consume more honey, it adds more pressure to assign workers quickly.

WorkTheNextShift tells each of the workers to do their jobs, then records how much honey gets consumed by the unassigned workers, and finally calls the UpdateStatusReport method.

Exercise Solution

```
static class HoneyVault
{
    private static decimal honey = Constants.INITIAL_HONEY;
    private static decimal nectar = Constants.INITIAL_NECTAR;

    internal static void Reset()
    {
        honey = Constants.INITIAL_HONEY;
        nectar = Constants.INITIAL_NECTAR;
    }

    public static bool ConsumeHoney(decimal amount)
    {
        if (honey >= amount)
        {
            honey -= amount;
            return true;
        }
        return false;
    }

    public static void CollectNectar(decimal amount)
    {
        if (amount > 0m) nectar += amount;
    }

    public static void ConvertNectarToHoney(decimal amount)
    {
        decimal nectarToConvert = amount;
        if (nectarToConvert > nectar) nectarToConvert = nectar;
        nectar -= nectarToConvert;
        honey += nectarToConvert * Constants.NECTAR_CONVERSION_RATIO;
    }

    public static string StatusReport
    {
        get
        {
            string status = $"{honey:0.00} units of honey\n" +
                            $"{nectar:0.00} units of nectar";
            string warnings = "";
            if (honey < Constants.LOW_LEVEL_WARNING) warnings +=
                                "\nLOW HONEY - ADD A HONEY MANUFACTURER";

            if (nectar < Constants.LOW_LEVEL_WARNING) warnings +=
                                "\nLOW NECTAR - ADD A NECTAR COLLECTOR";

            return status + warnings;
        }
    }
}
```

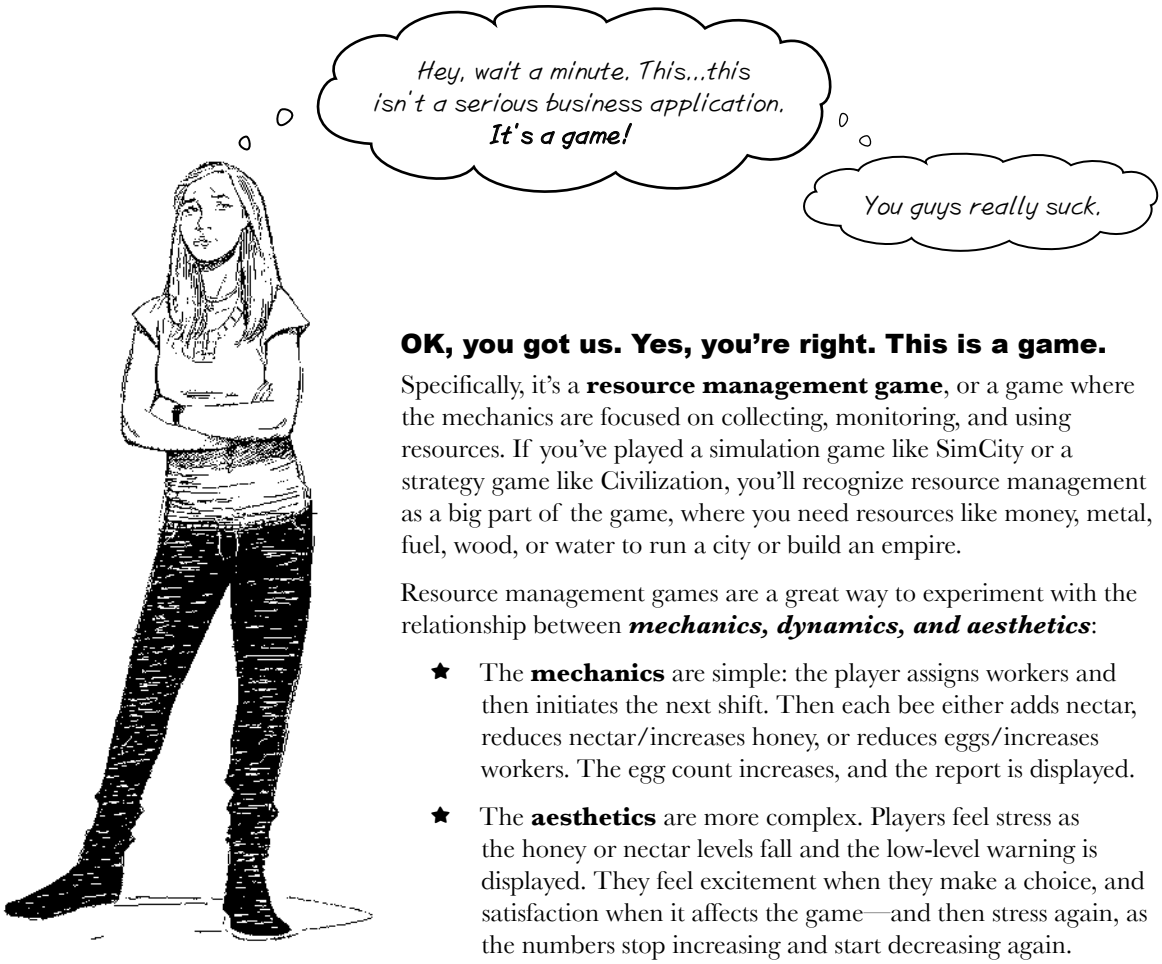
Here are the two constants that were added to the static Constants. They're used to initialize the private honey and nectar fields.

The ConsumeHoney method checks the amount parameter—if there's enough honey to do the job, it consumes it and returns true; otherwise, it returns false.

The NectarCollector bees use the CollectNectar method to add nectar to the hive. The if check is important so the bees don't accidentally add negative values and reduce the nectar.

The HoneyManufacturer bees use the ConvertNectarToHoney method when they're making honey. It reduces the nectar and adds the honey—but if they try to convert too much, it will only convert as much as is left in the vault.

The StatusReport property generates the honey vault portion of the status report, which the queen uses to generate the full status report. It also tells the screen reader to announce any warnings.



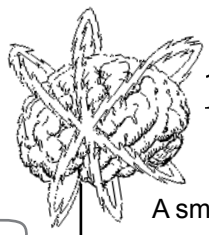
OK, you got us. Yes, you're right. This is a game.

Specifically, it's a **resource management game**, or a game where the mechanics are focused on collecting, monitoring, and using resources. If you've played a simulation game like SimCity or a strategy game like Civilization, you'll recognize resource management as a big part of the game, where you need resources like money, metal, fuel, wood, or water to run a city or build an empire.

Resource management games are a great way to experiment with the relationship between ***mechanics, dynamics, and aesthetics***:

- The **mechanics** are simple: the player assigns workers and then initiates the next shift. Then each bee either adds nectar, reduces nectar/increases honey, or reduces eggs/increases workers. The egg count increases, and the report is displayed.
- The **aesthetics** are more complex. Players feel stress as the honey or nectar levels fall and the low-level warning is displayed. They feel excitement when they make a choice, and satisfaction when it affects the game—and then stress again, as the numbers stop increasing and start decreasing again.
- The game is driven by the **dynamics**. There's nothing in the code that makes the honey or nectar scarce—they're just consumed by the bees and eggs.

Really take a minute and think about this, because it gets to the heart of what dynamics are about. Do you see any way to use some of these ideas in other kinds of programs, and not just games?



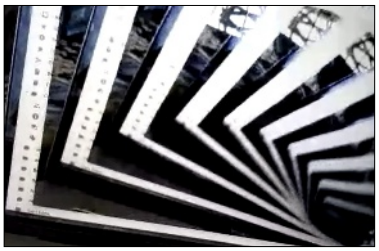
Brain Power

A small change in HoneyVault.NECTAR_CONVERSION_RATIO can make the game much easier or much harder by making the honey drain slowly or quickly. What other numbers affect gameplay? What do you think is driving those relationships?

Feedback drives your beehive management game

Let's take a few minutes and really understand how this game works. The nectar conversion ratio has a big impact on your game. If you change the constants, it can make really big differences in gameplay. If it takes just a little honey to convert an egg to a worker, the game gets really easy. If it takes a lot, the game gets much harder. But if you go through the classes, you won't find a difficulty setting. There's no Difficultly field on any of them. Your queen doesn't get special power-ups to help make the game easier, or tough enemies or boss battles to make it more difficult. In other words, there's ***no code that explicitly creates a relationship*** between the number of eggs or workers and the difficulty of the game. So what's going on?

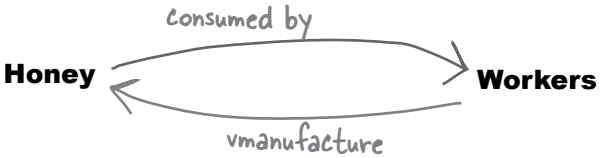
When you point a camera at a screen displaying its video output, you create a feedback loop that can cause these weird patterns.



You've probably played with **feedback** before. Start a video call between your phone and your computer. Hold the phone near the computer speaker and you'll hear noisy echoes. Point the camera at the computer screen and you'll see a picture of the screen inside the picture of the screen inside the picture of the screen, and it will turn into a crazy pattern if you tilt the phone. This is feedback: you're taking the live video or audio output and *feed*ing it right *back* into the input. There's nothing in the code of the video call app that specifically generates those crazy sounds or images. Instead, they **emerge** from the feedback.

Workers and honey are in a feedback loop

Your beehive management game is based on a series of **feedback loops**: lots of little cycles where parts of the game interact with each other. For example, honey manufacturers add honey to the vault, which is consumed by honey manufacturers, who make more honey.

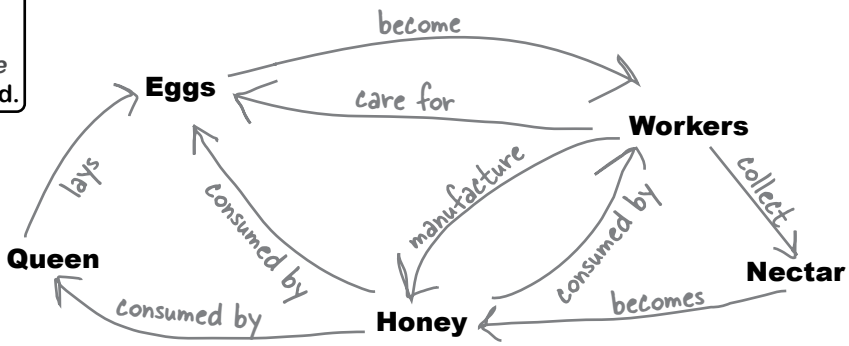


The feedback loop between the workers and honey is just one small part of the whole system that drives the game. See if you can spot it in the bigger picture that follows.

And that's just one feedback loop. There are many different feedback loops in your game, and they make the whole game more complex, more interesting, and (hopefully!) more fun.

A series of *feedback loops* drive the dynamics of your game. The code you build won't explicitly manage these feedback loops. They *emerge* out of the mechanics that you'll build.

And this same concept is actually really important in a lot of real-world business applications, not just games. Everything you're learning here, you can use on the job as a professional software developer.



Mechanics, Aesthetics, and Dynamics Up Close

Feedback loops…equilibrium…making your code do something indirectly by creating a system…does all this have your head spinning a little? Here's another opportunity to ***use game design to explore a larger programming concept.***

You've learned about mechanics, dynamics, and aesthetics—now it's time to bring them all together. The **Mechanics-Dynamics-Aesthetics framework**, or **MDA framework**, is a formal tool ("formal" just means it's written down) that's used by researchers and academics to analyze and understand games. It defines the relationship between mechanics, dynamics, and aesthetics, and gives us a way to talk about how they create feedback loops to influence each other.

The MDA framework was developed by Robin Hunicke, Marc LeBlanc, and Robert Zubek, and published in a 2004 paper called "MDA: A Formal Approach to Game Design and Game Research"—it's quite readable, without a ton of academic jargon. (Remember back in Chapter 5, when we talked about how aesthetics includes challenge, narrative, sensation, fantasy, and expression? That came from this paper.) Take a few minutes and look through it; it's actually a great read: *http://bit.ly/mda-paper*.

The goal of the MDA framework is to give us a formal way to think about and analyze video games. This may sound like something that's only important in an academic setting, like a college course on game design. But it's actually really valuable to all of us as developers, because it can help us understand how people perceive the games we create, and give us a deeper insight into **what makes those games fun**.

Game designers had been using the terms "mechanics," "dynamics," and "aesthetics" informally, but the paper really gave them a solid definition and established the relationship between them.

One thing that the MDA framework tackles is the **difference in perspective** between gamers and game designers. Players, first and foremost, want the game to be fun—but we've already seen how "fun" can differ wildly from player to player. Game designers, on the other hand, typically see a game through the lens of its mechanics, because they spend their time writing code, designing levels, creating graphics, and tinkering with the mechanical aspects of the game.

***All developers* (not just game developers!) can use the MDA framework to get a handle on feedback loops**

Let's use the MDA framework to analyze a classic game, Space Invaders, so we can better understand feedback loops:

- Start with the mechanics of the game: the player's ship moves left and right and fires shots up; the invaders march in formation and fire shots down; the shields block shots. The fewer enemies there are on-screen, the faster they go.
- Players figure out strategies: shoot where the invaders will be, pick off enemies on the sides of the formation, hide behind the shields. The code for the game doesn't have an `if/else` or `switch` statement for these strategies; they emerge as the player figures out the game. Players learn the rules, then start to understand the system, which helps them better take advantage of the rules. In other words, ***the mechanics and dynamics form a feedback loop***.
- The invaders get faster, the marching sounds speed up, and the player gets a rush of adrenaline. The game gets more exciting—and in turn, the player has to make decisions more quickly, causing them to make mistakes, which makes them change strategy, and that has an effect on the system. ***The dynamics and aesthetics form another feedback loop***.
- None of this happened by accident. The speed of the invaders, the rate at which they increase, the sounds, the graphics… these were all carefully balanced by the game's creator, Tomohiro Nishikado, who spent over a year designing it, drawing inspiration from earlier games, the writing of H. G. Wells, even his own dreams to create a classic game.

The Beehive Management System is turn-based... now let's convert it to real-time

A **turn-based game** is a game where the flow is broken down into parts—in the case of the Beehive Management System, into shifts. The next shift doesn't start until you click a button, so you can take all the time you want to assign workers. We can use a timer—like the one you used in Chapter 1—to **convert it to a real-time game** where time progresses continuously…and we can do it with just a few lines of code.

1 **Start a timer at the end of the MainPage constructor.**

Add this line of code to the end of the MainPage constructor to start a timer:

```
public MainPage()
{
    InitializeComponent();

    JobPicker.ItemsSource = new string[]
    {
        "Nectar Collector",
        "Honey Manufacturer",
        "Egg Care"
    };
    JobPicker.SelectedIndex = 0;

    UpdateStatusAndEnableAssignButton();

    Dispatcher.StartTimer(TimeSpan.FromSeconds(1.5), TimerTick);
}
```

You used a timer just like this in Chapter 1 to add a timer to your animal matching game. This code is very similar to the code you used in Chapter 1. Take a few minutes and flip back to that project to remind yourself how the timer works.

This statement starts a timer that calls the TimerTick method every 1.5 seconds.

2 **Generate the TimerTick method and call the Work Shift button event handler.**

We want the timer to keep the game moving forward, so we can have it automatically trigger the next shift if the player hasn't done it already. Use the IDE's Generate Method feature to generate the TimerTick method—notice that it generates it with a bool return type.

Here's the code for the method:

```
private bool TimerTick()
{
    if (!this.IsLoaded || !WorkShiftButton.IsVisible)
        return false;

    WorkShiftButton_Clicked(this, new EventArgs());
    return true;
}
```

If the TimerTick method returns false, the timer stops running. This `if` statement keeps the timer from trying to click the button if the page is in the process of closing, and also stops the timer when the hive runs out of honey.

Now run your game. A new shift starts every 1.5 seconds, whether or not you click the button. This is a small change to the mechanics, but it ***dramatically changes the dynamics of the game***, which leads to a huge difference in aesthetics. It's up to you to decide if the game is better as a turn-based or real-time simulation.

Curious about that EventArgs class? Learn more about it in our downloadable chapter on events and delegates.

*It took **just a few lines of code** to add the timer, but that completely changed the game. Is that because it had a big impact on the **relationship** between mechanics, dynamics, and aesthetics?*

Yes! The timer changed the mechanics, which altered the dynamics, which in turn impacted the aesthetics.

Let's take a minute and think about that feedback loop. The change in mechanics (a timer that automatically clicks the "Work the next shift" button every 1.5 seconds) creates a totally new dynamic: a window when players must make decisions, or else the game makes the decision for them. That increases the pressure, which gives some players a satisfying shot of adrenaline, but just causes stress in other players—the aesthetics changed, which makes the game more fun for some people but less fun for others.

There's a feedback loop here too. As players feel more stress, they make worse decisions, changing the game... aesthetics feeds back into mechanics.

But you only added half a dozen lines of code to your game, and none of them included "make this decision or else" logic. That's an example of behavior that **emerged** from how the timer and the button work together.

*This whole discussion of feedback loops seems pretty important, especially the part about **how behavior emerges**.*

Feedback loops and emergence are important programming concepts.

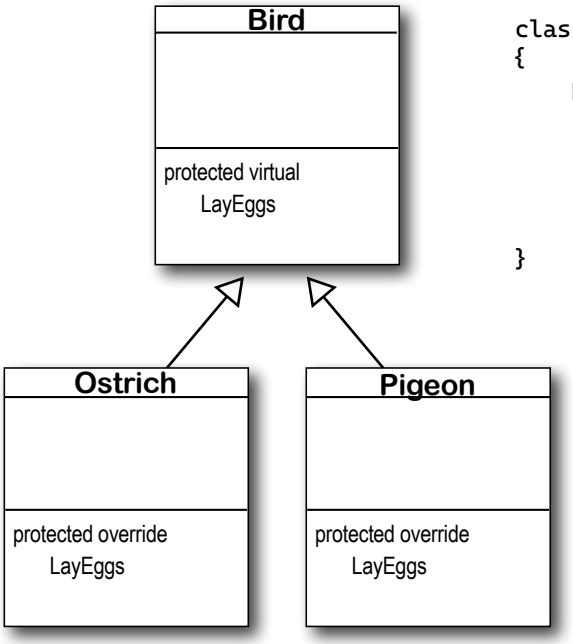
We designed this project to give you practice with inheritance, but *also* to let you explore and experiment with **emergent** behavior. That's behavior that comes not just from what your objects do individually, but also out of ***the way objects interact with each other***. The constants in the game (like the nectar conversion ratio) are an important part of that emergent interaction. When we created this exercise, we started out by setting those constants to some initial values, then we tweaked them by making tiny adjustments until we ended up with a system that's not quite in **equilibrium**—a state where everything is perfectly balanced—so the player needs to keep making decisions in order to make the game last as long as possible. That's all driven by the feedback loops between the eggs, workers, nectar, honey, and queen.

Try experimenting with these feedback loops. Add more eggs per shift or start the hive with more honey, for example, and the game gets easier. Go ahead, give it a try! You can change the entire feel of the game just by making small changes to a few constants.

Some classes should never be instantiated

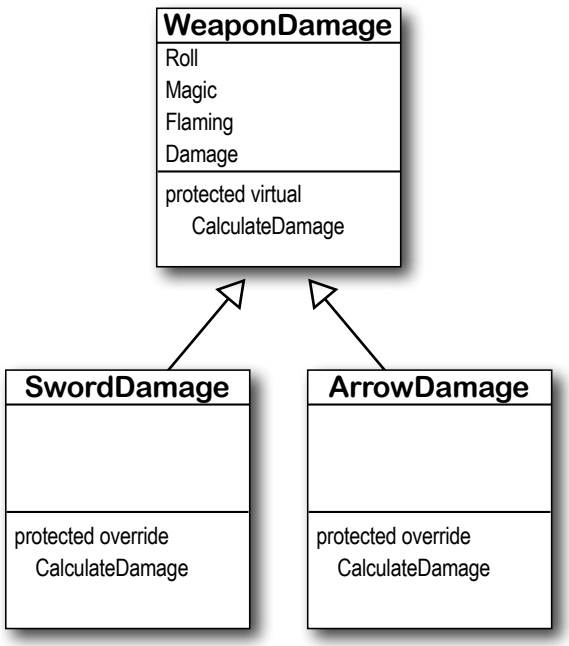
Remember our zoo simulator class hierarchy? You'll definitely end up instantiating a bunch of Hippos, Dogs, and Lions. What about the Canine and Feline classes? How about the Animal class? It turns out that **there are some classes that just don't need to be instantiated**…and, in fact, *don't make any sense* if they are.

Does that sound weird? Actually, it happens all the time—in fact, you created several classes earlier in the chapter that should never be instantiated.



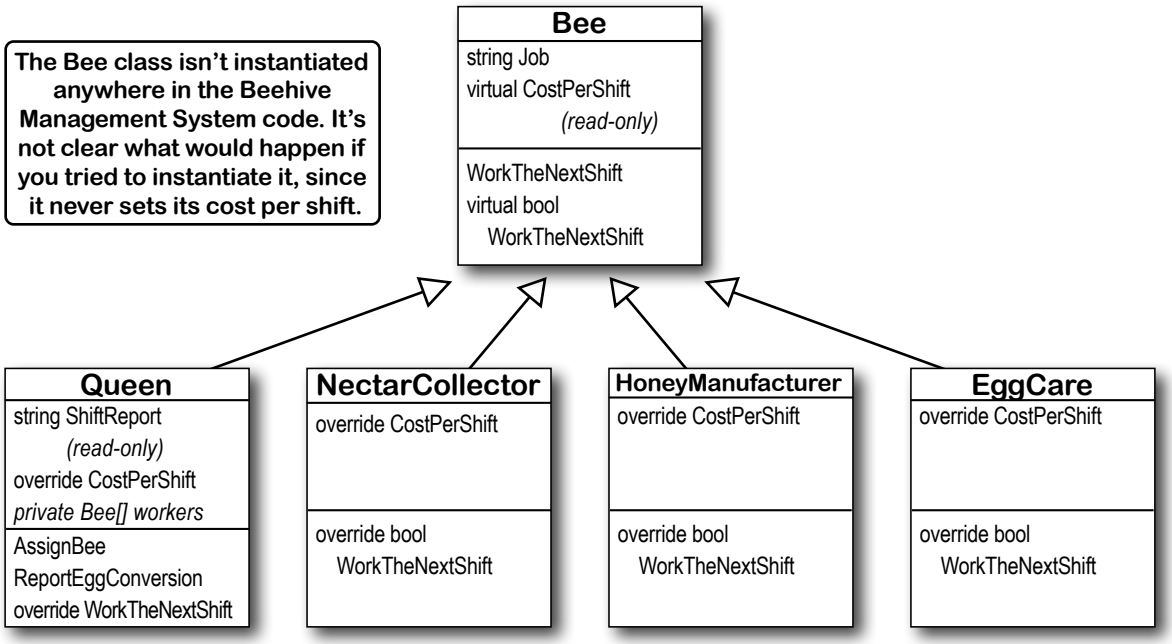

```
class Bird
{
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine
             ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```

Your Bird class was tiny—it just had a shared instance of Random and a LayEggs method that only existed so the subclasses could override it. Your WeaponDamage class was a lot bigger—it had a lot of properties. It also had a CalculateDamage class for the subclasses to override, which it called from its constructor.




```
class WeaponDamage
{
     /* ... code for the properties ... */ }

    protected virtual void CalculateDamage()
    {
        /* the subclass overrides this */
    }

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

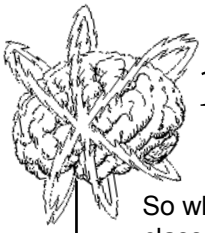
The Bee class isn't instantiated anywhere in the Beehive Management System code. It's not clear what would happen if you tried to instantiate it, since it never sets its cost per shift.




```
class Bee
{
    public virtual decimal CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public virtual bool WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
            return true;
        else
            return false;
    }
}
```

The Bee class had a WorkTheNextShift method that consumed honey and then did whatever job the bee was supposed to do—so it expected the subclass to override that method to actually do the job.

This method just consumes honey, but doesn't actually do a job. Why would we ever want to use it in our Beehive Management System?



Brain Power

So what happens when you instantiate the Bird, WeaponDamage, or Bee classes? Does it ever make sense to do it? Do all of their methods even work?

An abstract class is an intentionally incomplete class

It's really common to have a class with "placeholder" members that it expects the subclasses to implement. It could be at the top of the hierarchy (like your Bee, WeaponDamage, or Bird classes) or in the middle (like Feline or Canine in the zoo simulator class model). They take advantage of the fact that C# always calls the most specific method, like how WeaponDamage calls the CalculateDamage method that's only implemented in SwordDamage or ArrowDamage, or how Bee.WorkTheNextShift depends on the subclasses to override it.

C# has a tool that's built specifically for this: an **abstract class**. It's a class that's intentionally incomplete, with empty class members that serve as placeholders for the subclasses to implement. To make a class abstract, **add the `abstract` keyword to the class declaration**. Here's what you need to know about abstract classes.

- **An abstract class works just like a normal class.**
  You define an abstract class just like a normal one. It has fields and methods, and it can inherit from other classes, too, exactly like a normal class. There's almost nothing new to learn.

- **An abstract class can have incomplete "placeholder" members.**
  An abstract class can include declarations of properties and methods that must be implemented by inheriting classes. A method that has a declaration but no statements or method body is called an **abstract method**, and a property that only declares its accessors but doesn't define them is called an **abstract property**. Subclasses that extend it must implement all abstract methods and properties unless they're also abstract.

- **Only abstract classes can have abstract members.**
  If you put an abstract method or property into a class, then you'll have to mark that class abstract or your code won't compile. You'll learn more about how to mark a class abstract in a minute.

- **An abstract class can't be instantiated.**
  The opposite of abstract is **concrete**. A concrete method is one that has a body, and all the classes you've been working with so far are concrete classes. The biggest difference between an abstract class and a concrete class is that you can't use `new` to create an instance of an abstract class. If you do, C# will give you an error when you try to compile your code.

  Try it now! **Create a new console app**, add an empty abstract class, and try to instantiate it:

```
abstract class MyAbstractClass { }

MyAbstractClass myInstance = new MyAbstractClass();
```

  The compiler will give you an error, and won't let you build your code:

```
CS0144   Cannot create an instance of the abstract class or interface 'MyAbstractClass'
```

The compiler won't let you instantiate an abstract class because abstract classes are not meant to be instantiated.

Wait, what? A class that I can't instantiate? Why would I even want something like that?

Because you want to provide some of the code, but still require that subclasses fill in the rest of the code.

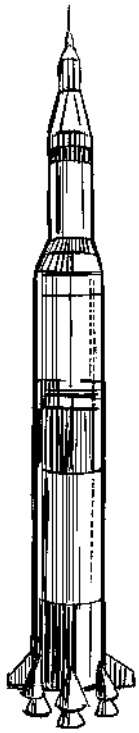
Sometimes ***bad things happen*** when you create objects that should never be instantiated. The class at the top of your class diagram usually has some fields that it expects its subclasses to set. An Animal class may have a calculation that depends on a Boolean called HasTail or Vertebrate, but there's no way for it to set that itself. ***Here's a quick example of a class that's problematic when instantiated…***

Let's plan a trip to another planet

Create a new console app, then **add the PlanetMission class and its two subclasses**, Mars and Venus.

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}

class Mars : PlanetMission
{
    public Mars()
    {
        kmToPlanet = 92000000;
        fuelPerKm = 1.73f;
        kmPerHour = 37000;
    }
}

class Venus : PlanetMission
{
    public Venus()
    {
        kmToPlanet = 41000000;
        fuelPerKm = 2.11f;
        kmPerHour = 29500;
    }
}
```

Do this!



Then **add these top-level statements** to instantiate each of the classes and call their MissionInfo methods:

```
Console.WriteLine(new Venus().MissionInfo());
Console.WriteLine(new Mars().MissionInfo());
Console.WriteLine(new PlanetMission().MissionInfo());
```

Before you run this code, can you figure out what it will print to the console?

Like we said, some classes should never be instantiated

Try running the PlanetMission console app. Did it do what you expected? It printed two lines to the console:

```
We'll burn 86509992 units of fuel in 1389 hours
We'll burn 159160000 units of fuel in 2486 hours
```

But then it threw an exception.

The problems all started when you created an instance of the PlanetMission class. Its FuelNeeded method expects the fields to be set by the subclass. When they aren't, they get their default values—zero. And when C# tries to divide a number by zero, this happens…

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

    3 references
    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}
```

Exception Unhandled

System.DivideByZeroException: 'Attempted to divide by zero.'

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings

Solution: use an abstract class

When you mark a class **abstract**, C# won't let you write code to instantiate it. So how does that fix this problem? It's like the old saying goes—prevention is better than cure. Add the **abstract** keyword to the PlanetMission class declaration:

```
abstract class PlanetMission
{
  // The rest of the class stays the same
}
```

As soon as you make the change, the compiler gives you an error:

CS0144 Cannot create an instance of the abstract class or interface 'PlanetMission'

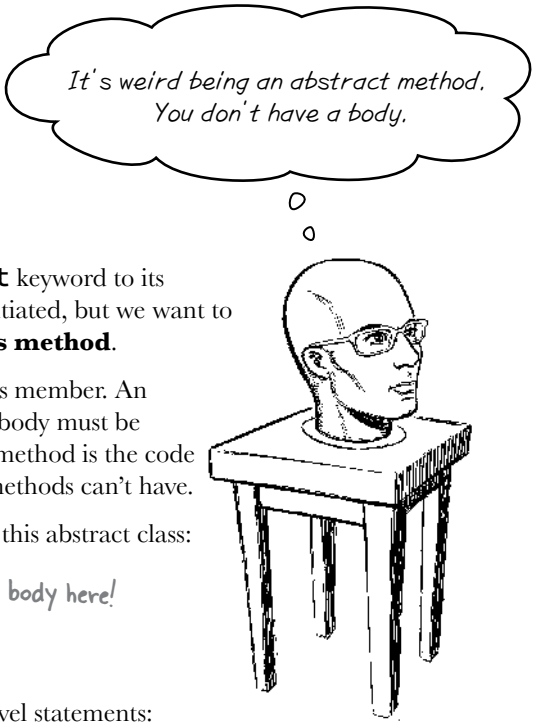
Your code won't compile at all—and no compiled code means no exception. This is really similar to the way you used the **private** keyword in Chapter 5, or **virtual** and **override** earlier in this chapter. Making some members private doesn't change the behavior. It just prevents your code from building if you break the encapsulation. The **abstract** keyword works the same way: you'll never get an exception instantiating an abstract class because the C# compiler *won't let you instantiate one in the first place*.

When you add the abstract keyword to a class declaration, the compiler gives you an error any time you try to create an instance of that class.

An abstract method doesn't have a body

The Bird class that you built was never meant to be instantiated. That's why it uses Console.Error to write an error message if a program tries to instantiate it and call its LayEggs method:

```
class Bird
{
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine
             ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```



Since we don't ever want to instantiate the Bird class, let's add the `abstract` keyword to its declaration. But that's not enough—not only should this class never be instantiated, but we want to **require** that every subclass that extends Bird **must override the LayEggs method**.

That's exactly what happens when you add the `abstract` keyword to a class member. An **abstract method** only has a class declaration but ***no method body***—the body must be implemented by any subclass that extends the abstract class. The **body** of a method is the code between the curly braces after the declaration—and it's something abstract methods can't have.

Go back to your Bird project from earlier and **replace the Bird class** with this abstract class:

```
abstract class Bird
{
    public abstract Egg[] LayEggs(int numberOfEggs);
}
```

No method body here!

Your program still runs like it did before! But try adding this line to the top-level statements:

```
Bird abstractBird = new Bird();
```

and you'll get a compiler error:

CS0144 Cannot create an instance of the abstract class or interface 'Bird'

Try to add a body to the LayEggs method:

```
public abstract Egg[] LayEggs(int numberOfEggs)
{
    return new Egg[0];
}
```

and you'll get a different compiler error:

CS0500 'Bird.LayEggs(int)' cannot declare a body because it is marked abstract

If an abstract class has abstract members, every subclass must override all of those members.

Abstract properties work just like abstract methods

Let's go back to the Bee class from our earlier example. We already know that we don't want the class to be instantiated, so let's modify it to turn it into an abstract class. We can do that just by adding the `abstract` modifier to the class declaration:

```
abstract class Bee
{
   /* the rest of the class stays the same */
}
```

But there's one other virtual member—and it's not a method. It's the CostPerShift property, which the Bee.WorkTheNextShift method calls to figure out how much honey the bee will require this shift:

```
public virtual decimal CostPerShift { get; }
```

We learned in Chapter 5 that properties are really just methods that are called like fields. Use the **`abstract` keyword to create an abstract property** just like you do with a method:

```
public abstract decimal CostPerShift { get; }
```

Abstract properties can have a get accessor, a set accessor, or both get and set accessors. Setters and getters in abstract properties **can't have method bodies**. Their declarations look just like automatic properties—but they're not, because they don't have any implementation at all. Like abstract methods, abstract properties are placeholders for properties that must be implemented by any subclass that extends their class.

Here's the whole abstract Bee class, complete with abstract method and property (but not the comments):

```
abstract class Bee
{
    public abstract decimal CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public virtual bool WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
            return true;
        else
            return false;
    }
}
```

Replace the Bee class in your Beehive Management System app with this new abstract one. It will still work! But now if you try to instantiate the Bee class with `new Bee();` you'll get a compiler error. Even more importantly, ***you'll get an error if you extend Bee but forget to implement CostPerShift***.

Exercise

It's time to get some practice with abstract classes—and you don't have to look far to find good candidates for classes to make abstract.

Earlier in the chapter, you modified your SwordDamage and ArrowDamage classes to extend a new class called WeaponDamage. Make the WeaponDamage class abstract. There's a good candidate for an abstract method in WeaponDamage—make that abstract as well.

there are no Dumb Questions

Q: When I mark a class abstract, does that change the way it behaves? Do the methods or properties work differently than they do in a concrete class?

A: No, abstract classes work exactly like any other kind of class. When you add the `abstract` keyword to the class declaration, it causes the C# compiler to do two things: prevent you from using the class in a `new` statement, and allow you to include abstract members.

Q: I'm confused about the way you're using the word "implement" or "implementation." What do you mean when you're talking about implementing an abstract method?

A: When you use the `abstract` keyword to declare an abstract method or property, we say that you're **defining** the abstract member. Later on, when you add a complete method or property with the same declaration to a concrete class, we say that you're **implementing** the member. So you define abstract methods or properties in an abstract class, and implement them in concrete classes that extend it.

Q: I'm still having trouble with the idea that the `abstract` keyword keeps my code from compiling if I try to instantiate an instance of an abstract class. I already have trouble finding and fixing all of the compiler errors. Why do I want to make it even harder to get my code to build?

A: Sometimes when you're first learning to code, those "CS" compiler errors can be a little frustrating. Everyone has spent time trying to track down a missing comma, period, or quotation mark to try to clear out the Errors List. So why would you ever use a keyword like `abstract` or `private` that puts even more restrictions on your code and makes those compiler errors even more common? It seems a little counterintuitive. If you never use the `abstract` keyword, you'll never see a "Cannot create an instance of the abstract class" compiler error. So why ever use it?

The reason you use keywords like `abstract` or `private` that keep your code from building in certain cases is that it's a lot easier to fix a "Cannot create an instance of the abstract class" compiler error than it is to track down the error that it prevents. If you have a class that should never be instantiated, it's because the bug you get when you create an instance of it instead of a subclass can be subtle and difficult to find. Adding `abstract` to the base class causes your code to **fail fast** with an error that's easier to fix.

Bugs caused by instantiating a base class that should never be instantiated can be subtle and hard to find. Making it abstract makes your code fail fast if you try to create an instance of it.

Exercise Solution

Thanks for refactoring this class! I bet you prevented some annoying bugs in the future. Now I can think about my game, and not code.
Great job!

The WeaponDamage class should never be instantiated—the only reason it exists is so that the SwordDamage and ArrowDamage classes can inherit its properties and methods. So it makes sense to mark this class abstract. Have a look at its CalculateDamage method:

```
protected virtual void CalculateDamage() {
  /* the subclass overrides this */
}
```

This method is a great candidate to convert to an abstract method, because it only exists so that subclasses will override it with their own implementations that update the Damage property. Here's everything that you needed to change in the WeaponDamage class:

```
abstract class WeaponDamage
{
    /* the Damage, Roll, Flaming, and Magic properties
       stay the same */

    protected abstract void CalculateDamage();

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

Was this the first time you've read through the code you wrote for previous exercises?

It may feel a little weird to go back to code you wrote before—but that's actually something a lot of developers do, and it's a habit you should get used to. Did you find things that you would do differently the second time around? Are there improvements or changes that you might make? It's always a good idea to take the time to refactor your code. That's exactly what you did in this exercise: you changed the structure of the code without modifying its behavior. ***That's refactoring.***

Inheritance is really useful. I can define a method once in a base class, and it automatically appears in each subclass. What if I want to do that for methods in two different classes? Is there a way for one subclass to ***extend two base classes****?*

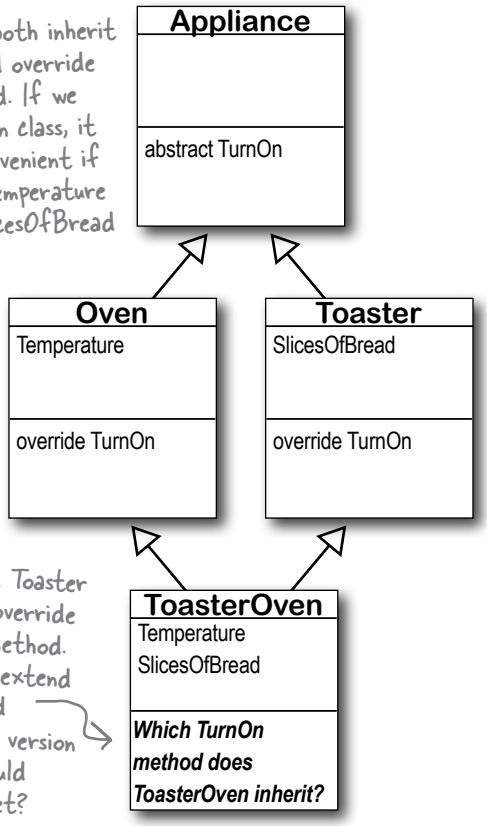
That sounds great! But there's a problem.

If C# let you inherit from more than one base class, it would open up a whole can of worms. When a language lets one subclass inherit from two base classes, it's called **multiple inheritance**. If C# supported multiple inheritance, you would end up in a big fat class conundrum called…

That's its real name! Some developers just call it the "diamond problem."

The Deadly Diamond of Death

Oven and Toaster both inherit from Appliance and override the TurnOn method. If we want a ToasterOven class, it would be really convenient if we could inherit Temperature from Oven and SlicesOfBread from Toaster.



The Oven and Toaster classes both override the TurnOn method. If C# let us extend both Oven and Toaster, which version of TurnOn would ToasterOven get?

What would happen in a CRAZY world where C# allowed multiple inheritance? Let's play a little game of "what if" and find out.

What if… you had a class called Appliance that had an abstract method called TurnOn?

And what if… it had two subclasses: Oven with a Temperature property, and Toaster with a SlicesOfBread property?

And what if… you wanted to create a ToasterOven class that inherited both Temperature and SlicesOfBread?

And what if… C# supported multiple inheritance, so you could do that?

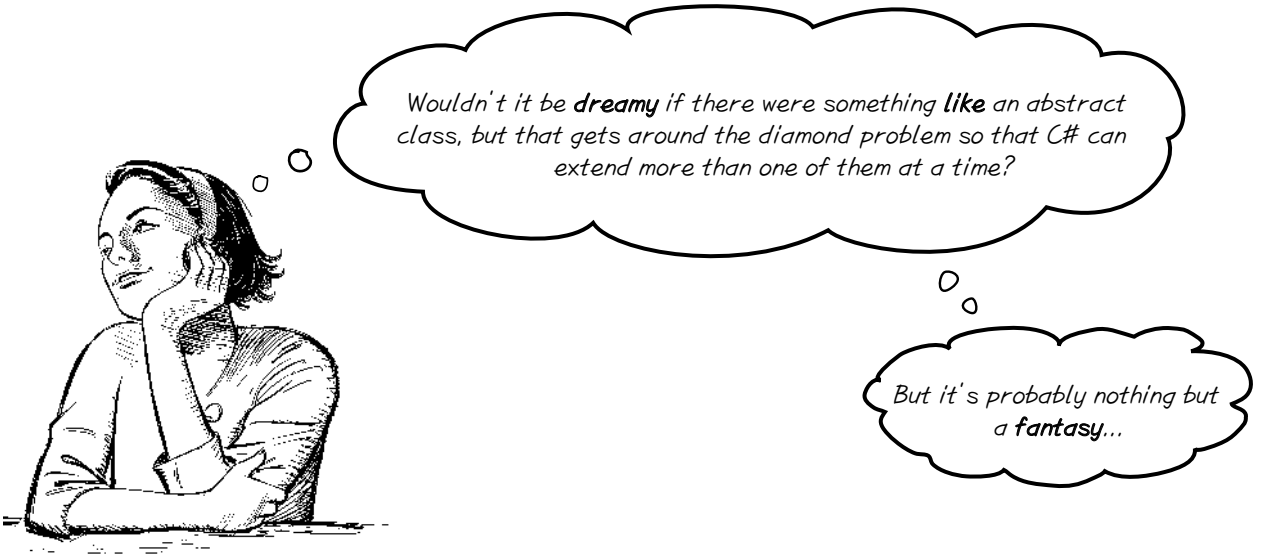
Then there's only one more question...

Which TurnOn does ToasterOven inherit?

Does it get the version from Oven? Or does it get the version from Toaster?

There's no way to know!

And that's why C# doesn't allow multiple inheritance.



Bullet Points

- A subclass can override members it inherits, replacing them with new methods or properties with the same names.
- To override a method or property, add the **`virtual` keyword** to the base class, then add the **`override` keyword** to the member with the same name in the subclass.
- The **`protected` keyword** is an access modifier that makes a member public only to its subclasses, but private to every other class.
- When a subclass overrides a method in its base class, the **more specific version** defined in the subclass is always called—even if the base class is calling it.
- If a subclass just adds a method with the same name as a method in its base class, it only **hides** the base class method instead of overriding it. Use the **`new` keyword** when you're hiding methods.
- The **dynamics** of a game describe how the mechanics combine and cooperate to drive the gameplay.
- A subclass can access its base class using the **`base` keyword**. When a base class has a constructor, your subclass needs to use the `base` keyword to call it.
- A subclass and base class can have **different constructors**. The subclass can choose what values to pass to the base class constructor.
- Build your **class model on paper** before you write code to help you understand and solve your problem.
- When your classes overlap as little as possible, that's an important design principle called **separation of concerns**.
- **Don't repeat yourself** (or **DRY**) is another design principle aimed at not repeating the code that does the same thing multiple places in an app's code.
- **Emergent behavior** occurs when objects interact with each other, beyond the logic directly coded into them.
- **Abstract classes** are intentionally incomplete classes that cannot be instantiated.
- Adding the **`abstract` keyword** to a method or property and leaving out the body makes it abstract. Any concrete subclass of the abstract class must implement it.
- **Refactoring** means reading code you already wrote and making improvements without modifying its behavior.
- C# does not allow for multiple inheritance because of the **diamond problem**: it can't determine which version of a member inherited from two base classes to use.

Unity Lab #3

GameObject Instances

C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

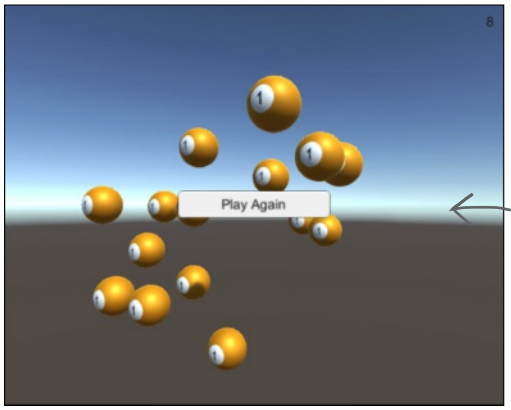
You've been creating objects in C# since you learned about the **`new`** keyword in Chapter 3. In this Unity Lab, you'll **create instances of a Unity GameObject** and use them in a complete, working game. This is a great starting point for writing Unity games in C#.

The goal of the next two Unity Labs is to **create a simple game** using the familiar billiard ball from the last lab. In this lab, you'll build on what you learned about C# objects and instances to start building the game. You'll use a **prefab**—Unity's tool for creating instances of GameObjects—to create lots of instances of a GameObject, and you'll use scripts to make your GameObjects fly around your game's 3D space.

Let's build a game in Unity!

Unity is all about building games. So in the next two Unity Labs, you'll use what you've learned about C# to build a simple game. Here's the game that you're going to create:

When you start the game, the scene slowly fills up with billiard balls. The player needs to keep clicking them to make them disappear. Once there are 15 balls in the scene, the game is over.

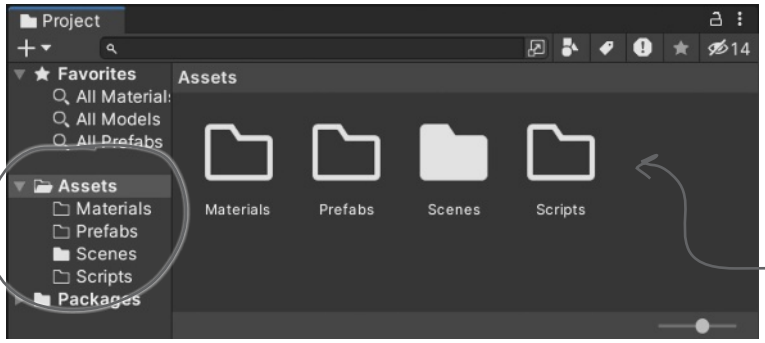


The game displays a score in the upper-right corner. Players score a point for each ball they click on.

When the game is over, a Play Again button lets the player start a new game.

So let's get started. The first thing you'll do is get your Unity project set up. This time we'll keep the files a little more organized, so you'll create separate folders for your materials and scripts—and one more folder for prefabs (which you'll learn about later in the lab):

1. Before you begin, close any Unity project that you have open. Also close Visual Studio—you'll let Unity open it for you.
2. **Create a new Unity project** using the 3D template, just like you did for the previous Unity Labs. Give it a name to help you remember which labs it goes with—name it ***Unity Labs 3 and 4***.
3. Choose the Wide layout so your screen matches the screenshots.
4. Create a folder for your materials underneath the Assets folder. **Right-click on the Assets folder** in the Project window and choose Create >> Folder. Name it ***Materials***.
5. Create another folder under Assets named ***Scripts***.
6. Create one more folder under Assets named ***Prefabs***.



Make sure you create the Materials, Scripts, and Prefabs folders underneath Assets.

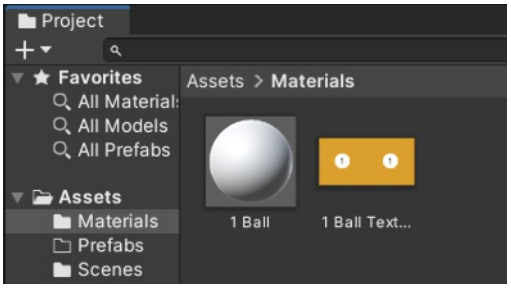
The Project window shows folders as outlines when they're empty.

Create a new material inside the Materials folder

Double-click on your new Materials folder to open it. You'll create a new material here.

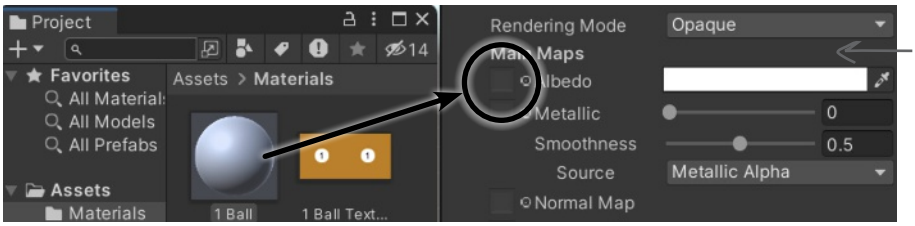
Go to *https://github.com/head-first-csharp/fifth-edition* and click on the Billiard Ball Textures link (just like you did in the first Unity lab) and download the texture file ***1 Ball Texture.png*** into a folder on your computer, then add it into your Materials folder—just like you did with the downloaded file in the first Unity Lab: right-click on the Materials folder you just created, then choose Import New Asset and add the file.

Now you can create the new material. Right-click on the Materials folder in the Project window and **choose Create >> Material**. Name your new material **1 Ball**. You should see it appear in the Materials folder in the Project window.



In the previous Unity Labs we used a *texture*, or a bitmap image file that Unity can wrap around GameObjects. When you dragged the texture onto a sphere, Unity automatically created a *material*, which is what Unity uses to keep track of information about how a GameObject should be rendered that can have a reference to a texture. This time you're creating the material manually. Just like last time, you may need to click the Download button on the GitHub page to download the texture PNG file.

Make sure the 1 Ball material is selected in the Materials window, so it shows up in the Inspector. Click on the *1 Ball Texture* file and **drag it into the box to the left of the Albedo label**.



Select the 1 Ball material in the Project window so you can see its properties, then drag the texture map onto the box to the left of the Albedo label.

You should now see a tiny little picture of the 1 Ball texture in the box to the left of Albedo in the Inspector.

Now your material looks like a billiard ball when wrapped around a sphere.

Behind the Scenes

GameObjects reflect light from their surfaces.

When you see an object in a Unity game with a color or texture map, you're seeing the surface of a GameObject reflecting light from the scene, and the **albedo** controls the color of that surface. "Albedo" is a term from physics (specifically astronomy) that means the color that's reflected by an object. You can learn more about albedo from the Unity Manual. Choose Unity Manual from the Help menu to open the manual in a browser and search for "albedo"—there's a manual page that explains albedo color and transparency.

Spawn a billiard ball at a random point in the scene

Create a new Sphere GameObject with a script called OneBallBehaviour:

- Choose 3D Object >> Sphere from the GameObject menu to **create a sphere**.
- Drag your new **1 Ball material** onto it to make it look like a billiard ball.
- Next, **right-click on the Scripts folder** that you created in the Project window, click Create in the menu, and choose C# Script. This will **create a new C# script** named OneBallBehaviour.
- **Drag the script onto the Sphere** in the Hierarchy window. Select the sphere and make sure a Script component called "One Ball Behaviour" shows up in the Inspector window.

Double-click on your new script to edit it in Visual Studio. ***Add exactly the same code*** that you used in BallBehaviour in the first Unity Lab, then **comment out the Debug.DrawRay line** in the Update method.

Your OneBallBehaviour class should now look like this:

Don't forget to drag your new script onto your sphere; otherwise, it won't spin.

```
public class OneBallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
        // Debug.DrawRay(Vector3.zero, axis, Color.yellow);
    }
}
```

When you add a *Start method* to a GameObject, Unity calls that method every time a new instance of that object is added to the scene. If the Start method is in a script attached to a GameObject that appears in the Hierarchy window, that method will get called *as soon as the game starts*.

Unity often instantiates a GameObject sometime before it's added to the scene. It only calls the Start method when the GameObject is actually added to the scene.

You won't need this line, so comment it out.

Now modify the Start method to move the sphere to a random position when it's created. You'll do this by setting **transform.position**, which changes the position of the GameObject in the scene. Here's the code to position your ball at a random point—**add it to the Start** method of your OneBallBehaviour script:

```
    // Start is called before the first frame update
    void Start()
    {
        transform.position = new Vector3(3 - Random.value * 6,
            3 - Random.value * 6, 3 - Random.value * 6);
    }
```

Remember, the Play button does not save your game! Make sure you save early and save often.

Use the Play button in Unity to run your game. A ball should now be circling the Y axis at a random point. Stop and start the game a few times. The ball should spawn at a different point in the scene each time.

Use the debugger to understand Random.value

You've used the Random class in several projects—specifically, its static Random.Shared property. You used it to scatter the animals in the animal matching game, pick random cards, and roll for damage. Unity's Random class is not the same class! Hover over the `Random` keyword in Visual Studio:

These classes are both called Random, but if you hover over them in Visual Studio to see the tooltip, you'll see that the one you used earlier is in the System namespace. Now you're using the Random class in the *UnityEngine namespace*.

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}
```

class UnityEngine.Random
Easily generate random data for games.

```
int
{
    int total = 0;
    for (int i = 0; i < numberOfRolls; i++) total += Random.Shared.Next(1, 7);
    return total;
}
```

class System.Random
Represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet certain statistical requirements for randomness.
GitHub Examples and Documentation

This is from the code you wrote earlier to roll for damage in Owen's app.

You can see from the code that this new Random class is different from the one you used before. Earlier you called Random.Next to get a random value, and that value was a whole number. This new code uses **Random.value**, but that's not a method—it's actually a property.

Unity may prompt you to enable debugging, just like in the last Unity Lab.

Use the Visual Studio debugger to see the kinds of values that this new Random class gives you. Click the "Attach to Unity" button (▶ Attach to Unity in Visual Studio, ▷ Attach to Unity in VSCode) to attach Visual Studio to Unity. Then **add a breakpoint** to the line you added to the Start method.

Now go back to Unity and **start your game**. It should break as soon as you press the Play button. Hover your cursor over `Random.value`—make sure it's over `value`. Visual Studio will show you its value in a tooltip:

```
8   void Start()
9   {
10      transform.position = new Vector3(3 - Random.value * 6,
11          3 - Random.value * 6, 3 - Random.value * 6);
12  }
13
```

Random.value 0.03689707

Keep Visual Studio attached to Unity and restart your game a few times. You'll get a new random number between 0 and 1 each time you restart it.

Keep Visual Studio attached to Unity, then go back to the Unity editor and **stop your game** (in the Unity editor, not in Visual Studio). Start your game again. Do it a few more times. You'll get a different random value each time. That's how UnityEngine.Random works: it gives you a new random value between 0 and 1 each time you access its value property.

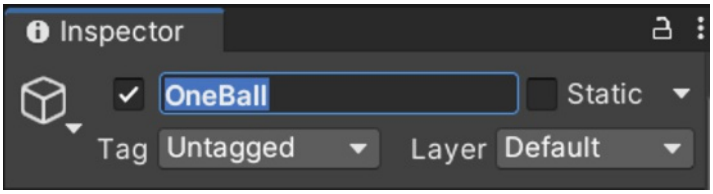
Press Continue (▶ Continue) to resume your game. It should keep running—the breakpoint was only in the **Start method**, which is ***just called once for each GameObject instance***, so it won't break again. Then go back to Unity and stop the game.

You can't edit scripts in Visual Studio while it's attached to Unity, so click the square Stop Debugging button to detach the Visual Studio debugger from Unity.

Turn your GameObject into a prefab

In Unity, a **prefab** is a GameObject that you can instantiate in your scene. Over the past few chapters, you've been working with object instances, and creating objects by instantiating classes. Unity lets you take advantage of objects and instances, so you can build games that reuse the same GameObjects over and over again. Let's turn your 1 ball GameObject into a prefab.

GameObjects have names. Change the name of your GameObject to *OneBall*. Start by **selecting your sphere**, by clicking on it in the Hierarchy window or in the scene. Then use the Inspector window to **change its name to OneBall**.



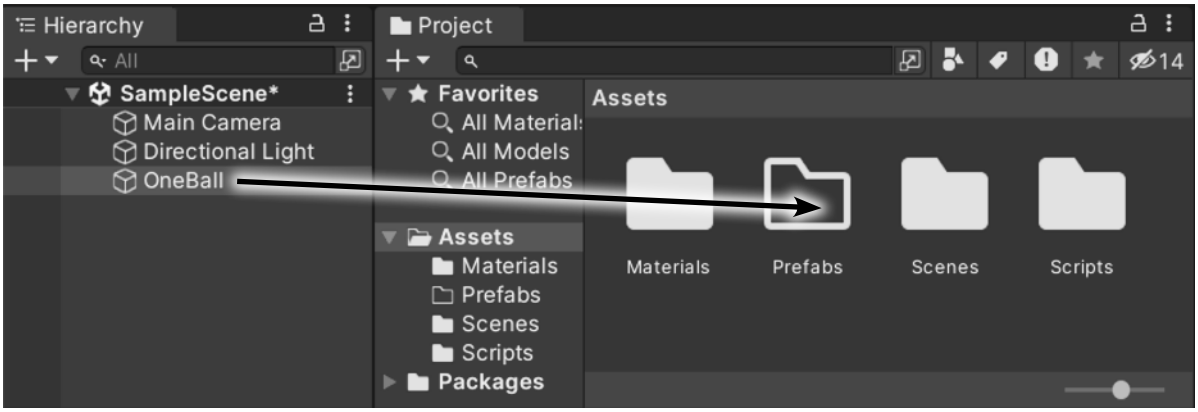
You can also rename a GameObject by right-clicking on it in the Hierarchy window and choosing Rename.

Watch it!

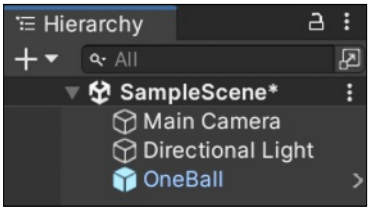
Visual Studio won't let you edit code while it's attached to Unity.

If you try to edit your code but find that Visual Studio won't let you make any changes, that means Visual Studio is probably still attached to Unity! Press the square Stop Debugging button to detach the debugger.

Now you can turn your GameObject into a prefab. **Drag OneBall from the Hierarchy window into the Prefabs folder**.



OneBall should now appear in your Prefabs folder. Notice that ***OneBall is now blue in the Hierarchy window***. This indicates that it's now a prefab—Unity turned it blue to tell you that an instance of a prefab is in your hierarchy. That's fine for some games, but for this game we want all of the instances of the balls to be created by scripts.



When a GameObject is blue in the Hierarchy window, Unity is telling you it's a prefab instance.

Right-click on OneBall in the Hierarchy window **and *delete* the OneBall GameObject from the scene**. You should now only see it in the Project window, and not in the Hierarchy window or the scene.

Have you been *saving your scene* as you go?
Save early, save often!

Create a script to control the game

The game needs a way to add balls to the scene (and eventually keep track of the score, and whether or not the game is over).

Right-click on the Scripts folder in the Project window and **create a new script called GameController**. Your new script will use two methods available in any GameObject script:

- **The Instantiate method creates a new instance of a GameObject.** When you're instantiating GameObjects in Unity, you don't typically use the `new` keyword like you saw in Chapter 2. Instead, you'll use the Instantiate method, which you'll call from the AddABall method.
- **The InvokeRepeating method calls another method in the script over and over again.** In this case, it will wait 1.5 seconds, then call the AddABall method once a second for the rest of the game.

Here's the source code for it:

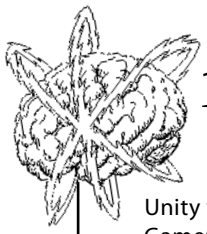
```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1);
    }

    void AddABall()
    {
        Instantiate(OneBallPrefab);
    }
}
```

What's the type of the second argument that you're passing to InvokeRepeating?

Unity's *InvokeRepeating method* calls another method over and over again. Its first parameter is a string with the name of the method to call ("invoke" just means calling a method).

This is a method called AddABall. All it does is create a new instance of a prefab.

You're passing the OneBallPrefab field as a parameter to the Instantiate method, which Unity will use to create an instance of your prefab.

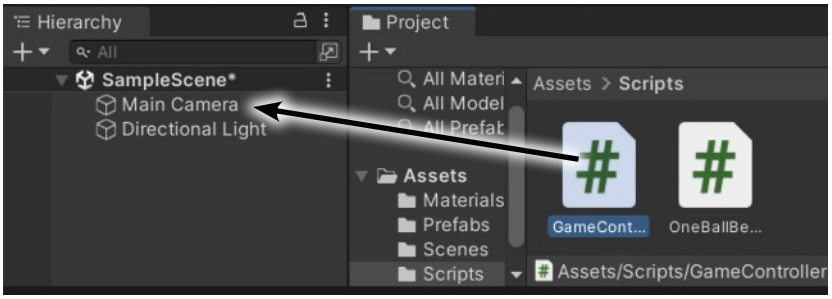


Brain Power

Unity will only run scripts that are attached to GameObjects in a scene. The GameController script will create instances of our OneBall prefab, but we need to attach it to something. Luckily, we already know that a camera is just a GameObject with a Camera component (and also an AudioListener). The Main Camera will always be available in the scene. So…what do you think you'll do with your new GameController script?

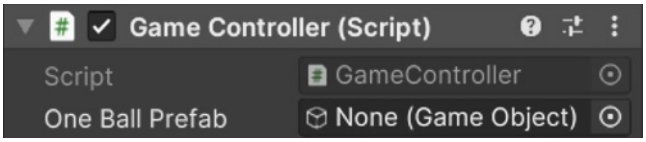
Attach the GameController script to the Main Camera

Your new GameController script needs to be attached to a GameObject to run. Luckily, the Main Camera is just another GameObject—it happens to be one with a Camera component and an AudioListener component—so let's attach your new script to it. **Drag your GameController** script out of the Scripts folder in the Project window and **onto the Main Camera** in the Hierarchy window.



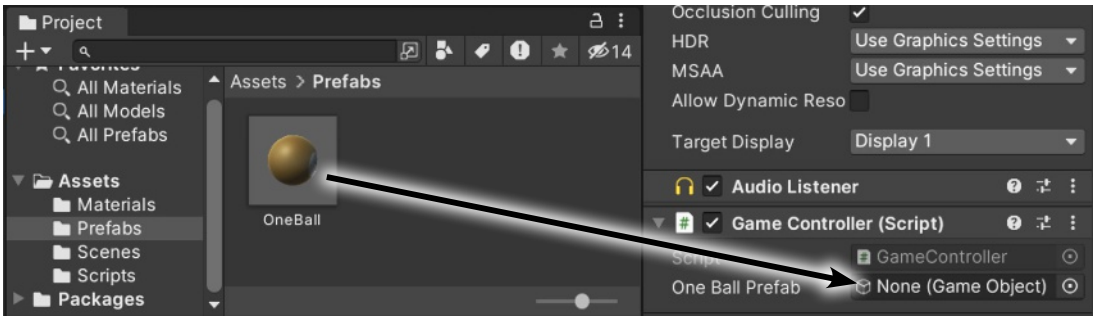
You learned about public vs. private fields in Chapter 5. When a script class has a public field, the Unity editor shows that field in the Script component in the Inspector. It adds spaces between uppercase letters to make its name easier to read.

Look in the Inspector—you'll see a component for the script, exactly like you would for any other GameObject. The script has a ***public field called OneBallPrefab***, so Unity displays it in the Script component.

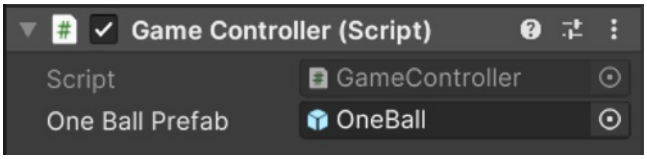


Here's the OneBallPrefab field in your GameController class. Unity added spaces before uppercase letters to make it easier to read (just like we saw in the last lab).

The OneBallPrefab field still says None, so we need to set it. **Drag OneBall out of the Prefabs folder** and onto the **box next to the One Ball Prefab label**.



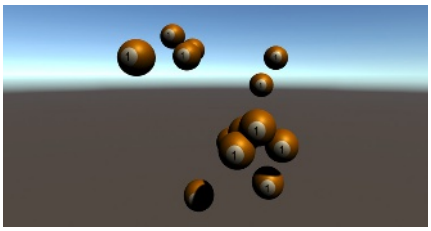
Now the GameController's OneBallPrefab field contains a ***reference*** to the OneBall prefab:



Go back to the code and **look closely at the AddABall method**. It calls the Instantiate method, passing it the OneBallPrefab field as an argument. You just set that field so that it contains your prefab. So every time GameController calls its AddABall method, it will ***create a new instance of the OneBall prefab***.

Press Play to run your code

Your game is all ready to run. The GameController script attached to the Main Camera will wait 1.5 seconds, then instantiate a OneBall prefab every second. Each instantiated OneBall's Start method will move it to a random position in the scene, and its Update method will rotate it around the Y axis every 2 seconds using OneBallBehaviour fields (just like in the last Lab). Watch as the play area slowly fills up with rotating balls:

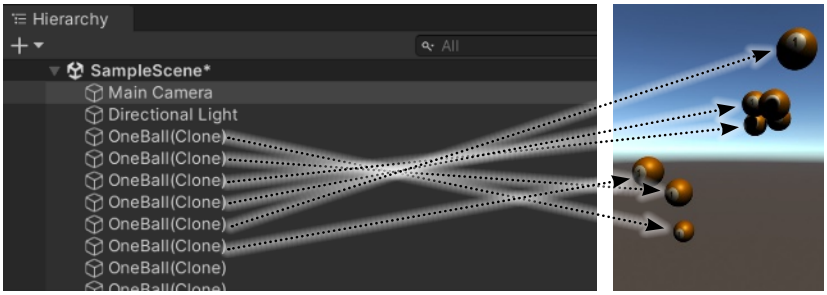


Unity calls every GameObject's Update method before each frame. That's called the *update loop*.

When you instantiate GameObjects in your code, they show up in the Hierarchy window when you run your game.

Watch the live instances in the Hierarchy window

Each of the balls flying around the scene is an instance of the OneBall prefab. Each of the instances has its own instance of the OneBallBehaviour class. You can use the Hierarchy window to track all of the OneBall instances—as each one is created, a "OneBall(Clone)" entry is added to the Hierarchy.



Click on any of the OneBall(Clone) items to view it in the Inspector. You'll see its Transform values change as it rotates, just like in the last lab.

We've included some coding exercises in the Unity Labs. They're just like the exercises in the rest of the book—and remember, it's not cheating to peek at the solution.

Exercise

Figure out how to add a BallNumber field to your OneBallBehaviour script so that when you click on a OneBall instance in the Hierarchy and check its One Ball Behaviour (Script) component, under the X Rotation, Y Rotation, Z Rotation, and Degrees Per Second labels it has a Ball Number field:

The first instance of OneBall should have its Ball Number field set to 1. The second instance should have it set to 2, the third 3, etc. ***Here's a hint:*** *You'll need a way to keep track of the count that's shared by all of the OneBall instances. You'll modify the Start method to increment it, then use it to set the BallNumber field.*

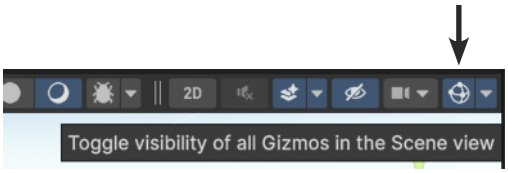
Use the Inspector to work with GameObject instances

Run your game. Once a few balls have been instantiated, click the Pause button—the Unity editor will jump back to the Scene view. Click one of the OneBall instances in the Hierarchy window to select it. The Unity editor will outline it in the Scene window to show you which object you selected. Go to the Transform component in the Inspector window and **set its Z scale value to 4** to make the ball stretch.

Start your simulation again—now you can track which ball you're modifying. Try changing its DegreesPerSecond, XRotation, YRotation, and ZRotation fields like you did in the last lab.

While the game is running, switch between the Game and Scene views. You can use the Gizmos in the Scene view ***while the game is running***, even for GameObject instances that were created using the Instantiate method (rather than added to the Hierarchy window).

Try clicking the Gizmos button at the top of the toolbar to toggle them on and off. You can turn on the Gizmos in the Game view, and you can turn them off in the Scene view.



Exercise Solution

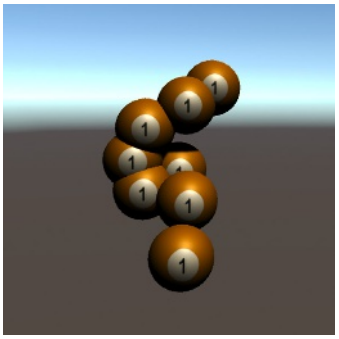
You can add a BallNumber field to the OneBallBehaviour script by keeping track of the total number of balls added so far in a static field (which we called BallCount). Each time a new ball is instantiated, Unity calls its Start method, so you can increment the static BallCount field and assign its value to that instance's BallNumber field.

```
static int BallCount = 0;
public int BallNumber;

// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);

    BallCount++;
    BallNumber = BallCount;
}
```

All of the OneBall instances share a single static BallCount field, so the first instance's Start method increments it to 1, the second instance increments BallCount to 2, the third increments it to 3, etc.

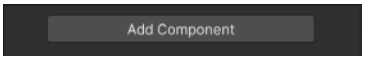
Use physics to keep balls from overlapping

Did you notice that occasionally some of the balls overlap each other?



Unity has a powerful **physics engine** that you can use to make your GameObjects behave like they're real, solid bodies—and one thing that solid shapes don't do is overlap each other. To prevent that overlap, you just need to tell Unity that your OneBall prefab is a solid object.

Stop your game, then **click on the OneBall prefab in the Project window** to select it. Then go to the Inspector and scroll all the way down to the bottom to the Add Component button:



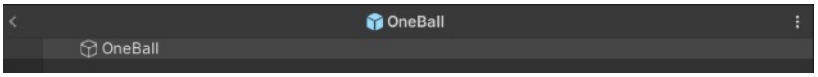
Click the button to pop up the Component window. **Choose Physics** to view the physics components, then **select Rigidbody** to add the component.

Make sure that you uncheck Use Gravity. Otherwise the balls will all react to gravity and start falling, and since there's nothing for them to hit, they'll keep falling forever.

While you're running physics experiments, here's one Galileo would appreciate. Try checking the Use Gravity box while your game is running. New balls that get created will start falling, occasionally hitting another ball and knocking it out of the way.

Run your game again—now you won't see balls overlap. Occasionally one ball will get created on top of another one. When that happens, the new ball will knock the old one out of the way.

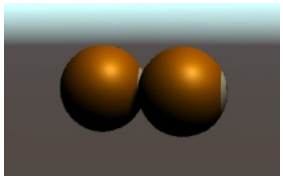
Let's run a little physics experiment to prove that the balls really are rigid now. Start your game, then pause it as soon as there are more than two balls created. Go to the Hierarchy window. If it looks like this:



then you're editing the prefab—click the back caret (<) in the top-right corner of the Hierarchy window to get back to the scene (you may need to expand SampleScene again).

You can use the Hierarchy window to delete GameObjects from your scene while the game is running.

- Hold down the Shift key, click the first OneBall instance in the Hierarchy window, and then click the second one so the first two OneBall instances are selected.
- You'll see dashes (—) in the Position boxes in the Transform panel. **Set the Position to (0, 0, 0)** to set both OneBall instances' positions at the same time.
- Use Shift-click to select any other instances of OneBall, right-click, and **choose Delete** to delete them from the scene so only the two overlapping balls are left.
- Unpause your game—the balls can't overlap now, so instead they'll be rotating next to each other.



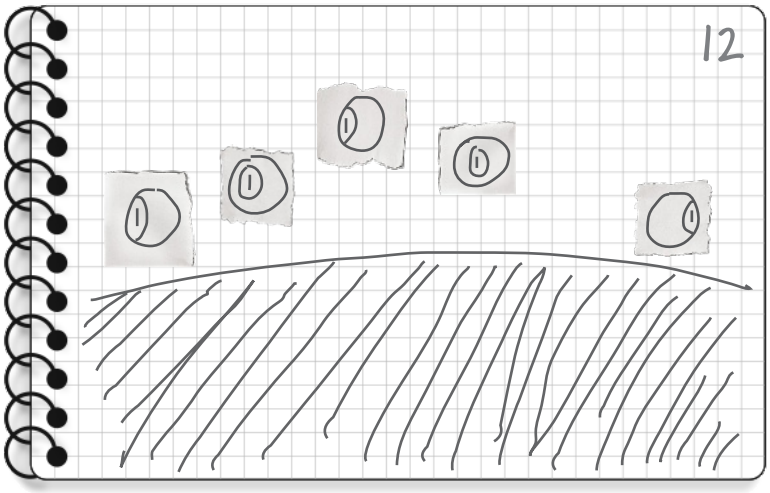
Stop the game in Unity and Visual Studio and *save your scene*.

Get creative!

You're halfway done with the game! You'll finish it in the next Unity Lab. In the meantime, this is a great opportunity to practice your **paper prototyping** skills. We gave you a description of the game at the beginning of this Unity Lab. Try creating a paper prototype of the game. Can you come up with ways to make it more interesting?

Draw the background of the Unity scene on a piece of paper, then draw billiard balls on scraps of paper.

Can you come up with a way to use the 8 ball from the first two Unity Labs to make the game more fun?



Here's the Play Again button that's displayed when the game is over.

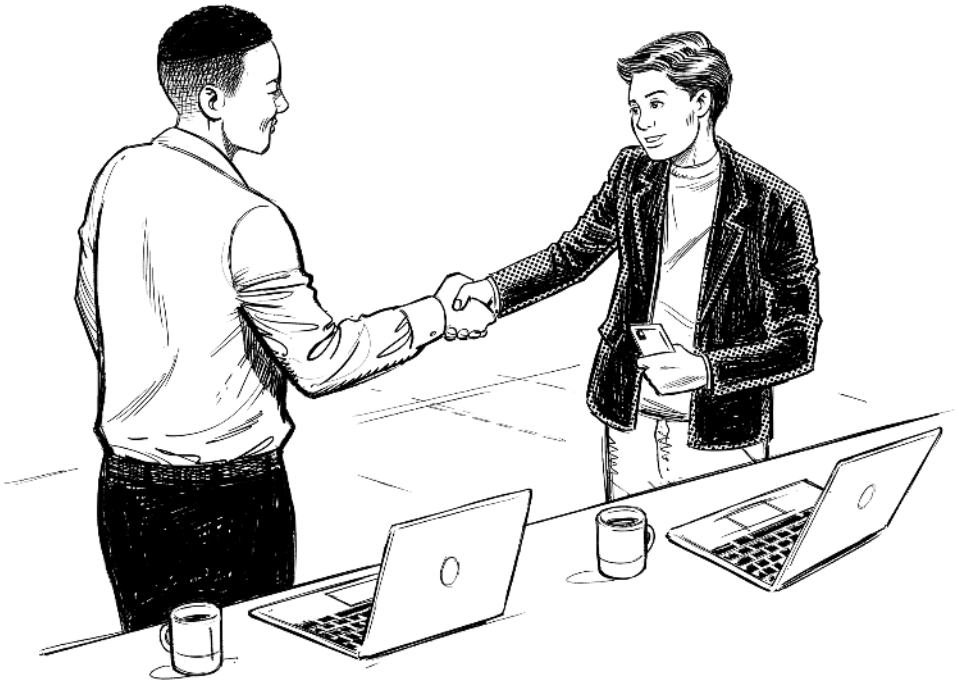
PLAY AGAIN

Bullet Points

- **"Albedo"** is a physics term that means the color that's reflected by an object. Unity can use texture maps for the albedo in a material.
- Unity has its own **Random class** in the UnityEngine namespace. The static Random.value method returns a random number between 0 and 1.
- A **prefab** is a GameObject that you can instantiate in your scene. You can turn any GameObject into a prefab.
- The **Instantiate method** creates a new instance of a GameObject. The Destroy method destroys it. Instances are created and destroyed at the end of the update loop.
- The **InvokeRepeating method** calls another method in the script over and over again.
- Unity calls every GameObject's Update method before each frame. That's called the **update loop**.
- You can **inspect the live instances** of your prefabs by clicking on them in the Hierarchy window.
- When you add a **Rigidbody** component to a GameObject, Unity's physics engine makes it act like a real, solid, physical object.
- The Rigidbody component lets you turn **gravity** on or off for a GameObject.

7 interfaces, casting, and "is"

Making classes keep their promises

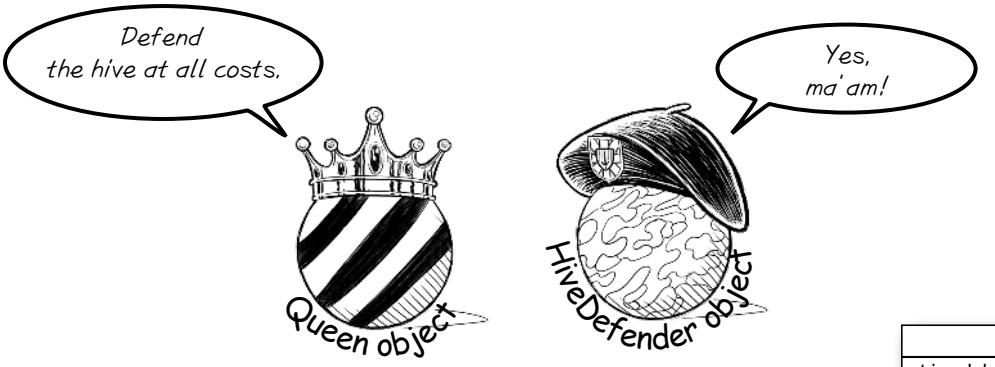


Need an object to do a specific job? Use an interface.

Sometimes you need to group your objects together based on the things they can do rather than the classes they inherit from—and that's where interfaces come in. You can use an interface to define a **specific job**. Any instance of a class that **implements** the interface is *guaranteed to do that job*, no matter what other classes it's related to. To make it all work, any class that implements an interface must promise to fulfill all of its obligations…or the compiler will break its kneecaps, see?

The beehive is under attack!

An enemy hive is trying to take over the queen's territory and keeps sending enemy bees to attack her workers. She's added a new elite Bee subclass called HiveDefender to help defend the hive.

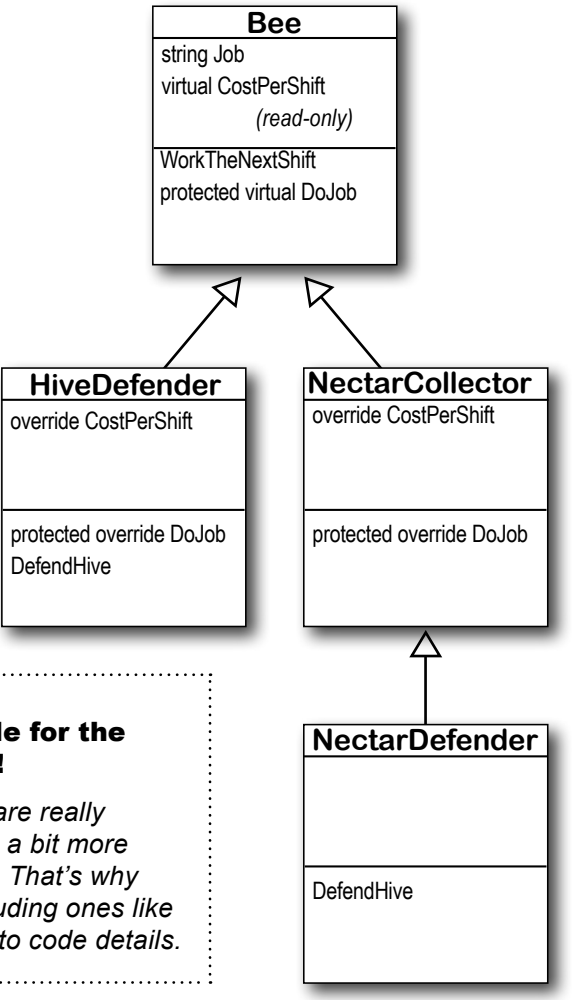


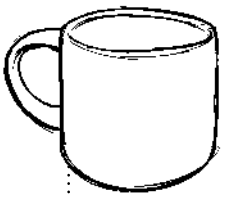
HiveDefender needs a DefendHive method because enemies can attack at any time

We can add a HiveDefender subclass to the Bee class hierarchy by extending the Bee class, overriding its CostPerShift with the amount of honey each defender consumes every shift, and overriding the DoJob method to fly out to the enemy hive and attack the enemy bees.

But enemy bees can attack at any time. We want defenders to be able to defend the hive ***whether or not they're currently doing their normal jobs***.

So in addition to DoJob, we'll add a DefendHive method to any Bee that can defend the hive—not just the elite HiveDefender workers, but any of their sisters who can take up arms and protect their queen. The queen will call her workers' DefendHive methods any time she sees that her hive is under attack.





Relax

We didn't include the complete code for the HiveDefender class...and that's OK!

This chapter is all about interfaces, which are really useful and very important! But they're also a bit more abstract than other C# concepts we've covered so far. That's why we've included ***lots of examples*** *in this chapter—including ones like this that show an idea without getting bogged down into code details.*

We could use casting to call the DefendHive method...

When you coded the Queen.DoJob method, you used a `foreach` loop to get each Bee reference in the `workers` array, then you used that reference to call worker.DoJob. If the hive is under attack, the queen will want to call her defenders' DefendHive methods. So we'll give her a HiveUnderAttack method that gets called any time the hive is being attacked by enemy bees, and she'll use a `foreach` loop to order her workers to defend the hive until all of the attackers are gone.

But there's a problem. The queen can use the Bee references to call DoJob because each subclass overrides Bee.DoJob, but she can't use a Bee reference to call the DefendHive method, because that method isn't part of the Bee class. So how does she call DefendHive?

Since DefendHive is only defined in each subclass, we'll need to use **casting** to convert the Bee reference to the correct subclass in order to call its DefendHive method.

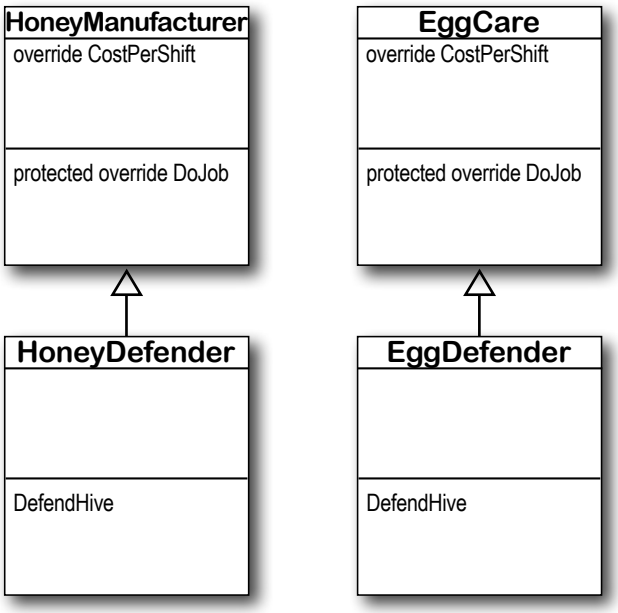
```
public void HiveUnderAttack() {
    foreach (Bee worker in workers) {
        if (EnemyHive.AttackingBees > 0) {
            if (worker.Job == "Hive Defender") {
                HiveDefender defender = (HiveDefender) worker;
                defender.DefendHive();
            } else if (worker.Job == "Nectar Defender") {
                NectarDefender defender = (NectarDefender) defender;
                defender.DefendHive();
            }
        }
    }
}
```

...but what if we add more Bee subclasses that can defend?

Some honey manufacturer and egg care bees want to step up and defend the hive, too. That means we'll need to add more `else` blocks to her HiveUnderAttack method.

This is getting complicated! The Queen.DoJob method is nice and simple—a very short `foreach` loop that takes advantage of the Bee class model to call the specific version of the DoJob method that was implemented in the subclass. We can't do that with DefendHive because it's not part of the Bee class—and we don't want to add it, because not all bees can defend the hive. ***Is there a better way to have unrelated classes do the same job?***

The honey manufacturer and egg care bees want to help defend the hive too. Does the queen need to have an if/else for each subclass?



An interface defines methods and properties that a class must implement...

An **interface** works just like an abstract class: you use abstract methods, and then you use the colon (:) to make a class implement that interface.

So if we wanted to add defenders to the hive, we could have an interface called IDefender. Here's what that looks like. It uses the **`interface` keyword** to define the interface, and it includes a single member, an abstract method called Defend. All members in an interface are public and abstract by default, so C# keeps things simple by having you ***leave off the* `public` *and* `abstract` *keywords***:

```
interface IDefender
{
    void Defend();
}
```

This interface has one member, a public abstract method called Defend. This works just like the abstract methods you saw in Chapter 6.

Any class that implements the IDefender interface **must include a Defend method** whose declaration matches the one in the interface. If it doesn't, the compiler will give an error.

...but there's no limit to the number of interfaces a class can implement

We just said that you use a colon (:) to make a class implement an interface. What if that class is already using a colon to extend a base class? No problem! **A class can implement many different interfaces, even if it already extends a base class:**

```
class NectarDefender : NectarCollector, IDefender
{
   void Defend() {
       /* Code to defend the hive */
   }
}
```

Since the Defend method is part of the IDefend interface, the NectarDefender class must implement it or it won't compile.

Now we have a class that can act like a NectarCollector, but can also defend the hive. NectarCollector extends Bee, so if you **use it from a Bee reference** it acts like a Bee:

```
Bee worker = new NectarDefender();
Console.WriteLine(worker.Job);
worker.WorkTheNextShift();
```

But if you **use it from an IDefender reference**, it acts like a hive defender:

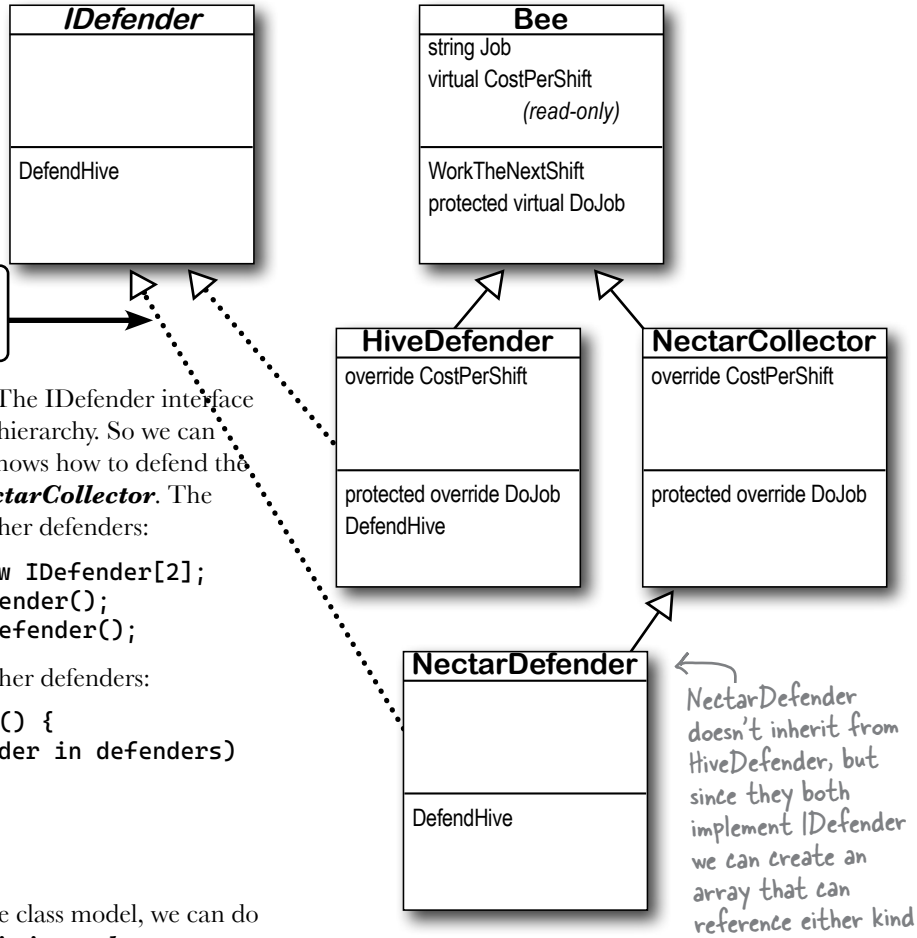
```
IDefender defender = new NectarCollector();
defender.Defend();
```

When a class implements an interface, it must include all of the methods and properties listed inside the interface or the code won't build.

Interfaces let unrelated classes do the same job

Interfaces can be a really powerful tool to help you design C# code that's easy to understand and build. Start by thinking about **specific jobs that classes need to do**, because that's what interfaces are all about.

Any Bee can implement the IDefender interface, no matter where it is in the class hierarchy. As long as it has a DefendHive method, the code will build.



We used *dotted lines* in the class diagram to show that an interface was implemented.

So how does this help the queen? The IDefender interface lives entirely outside the Bee class hierarchy. So we can add a NectarDefender class that knows how to defend the hive, and ***it can still extend NectarCollector***. The queen can keep an array of all of her defenders:

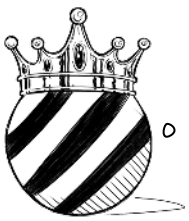
```
IDefender[] defenders = new IDefender[2];
defenders[0] = new HiveDefender();
defenders[1] = new NectarDefender();
```

That makes it easy for her to rally her defenders:

```
private void DefendTheHive() {
  foreach (IDefender defender in defenders)
  {
     defender.Defend();
  }
}
```

And since it lives outside of the Bee class model, we can do this ***without modifying any existing code***.

NectarDefender doesn't inherit from HiveDefender, but since they both implement IDefender we can create an array that can reference either kind of object.



Now that I know you can defend the hive, we'll all be a lot safer!

Relax

We'll give you *lots* of examples.

Still a little puzzled about how interfaces work and why you would use them? Don't worry—that's normal! The syntax is pretty straightforward, but there's a lot of subtlety. So we'll spend more time on interfaces…and we'll give you plenty of examples, and lots of practice.

Get a little practice using interfaces

The best way to understand interfaces is to start using them. Go ahead and **create a new Console App** project.

Do this!

1. **Add the top-level statements and TallGuy class.** Here's the code for a class called TallGuy and top-level statements that first instantiate it and then call its TalkAboutYourself method:

```
TallGuy tallGuy = new TallGuy("Jimmy", 76);
tallGuy.TalkAboutYourself();

class TallGuy
{
    private string? name;
    private int height;

    public TallGuy(string? name, int height)
    {
        this.name = name;
        this.height = height;
    }

    public void TalkAboutYourself()
    {
        Console.WriteLine($"My name is {name} and I'm {height} inches tall.");
    }
}
```

2. **Add an interface.** We're going to make TallGuy implement an interface. Add a new IClown interface to your project: right-click on the project in the Solution Explorer, **select Add >> New Item (Visual Studio) or Add New File (VSCode), and choose Interface**. Make sure it's called *IClown.cs*. The IDE will create an interface that includes the interface declaration. Add a Honk method:

```
interface IClown
{
    void Honk();
}
```

You don't need to add "public" or "abstract" inside the interface, because it automatically makes every property and method public and abstract.

3. **Try coding the rest of the IClown interface.** Before you go on to the next step, see if you can create the rest of the IClown interface, and modify the TallGuy class to implement this interface. In addition to the void method called Honk that doesn't take any parameters, your IClown interface should also have a read-only string property called FunnyThingIHave that has a get accessor but no set accessor.

Interface names start with *I*

Whenever you create an interface, you should make its name start with an uppercase **I**. There's no rule that says you need to do it, but it makes your code a lot easier to understand. You can see for yourself just how much easier that can make your life. Just go into the IDE to any blank line inside any method and type "I"—IntelliSense shows .NET interfaces.

4 **Here's the IClown interface.** Did you get it right? It's OK if you put the Honk method first—the order of the members doesn't matter in an interface, just like it doesn't matter in a class.

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

The IClown interface requires any class that implements it to have a void method called Honk and a string property called FunnyThingIHave that has a get accessor.

5 **Modify the TallGuy class so that it implements IClown.** Remember, the colon operator is always followed by the base class to inherit from (if any), and then a list of interfaces to implement, all separated by commas. Since there's no base class and only one interface to implement, the declaration looks like this:

```
class TallGuy : IClown
```

Then make sure the rest of the class is the same, including the two fields and the method. Select Build Solution from the Build menu in the IDE to compile and build the program. You'll see two errors:

```
CS0535  'TallGuy' does not implement interface member 'IClown.FunnyThingIHave'
CS0535  'TallGuy' does not implement interface member 'IClown.Honk()'
```

6 **Fix the errors by adding the missing interface members.** The errors will go away as soon as you add all of the methods and properties defined in the interface. So go ahead and implement the interface. Add a read-only string property called FunnyThingIHave with a get accessor that always returns the string "big shoes". Then add a Honk method that writes "Honk honk!" to the console.

Here's what it'll look like:

```
public string FunnyThingIHave
{
    get { return "big shoes"; }
}

public void Honk()
{
    Console.WriteLine("Honk honk!");
}
```

Any class that implements the IClown interface *must have* a void method called Honk and a string property called FunnyThingIHave that has a get accessor. The FunnyThingIHave property is allowed to have a set accessor too. The interface doesn't specify it, so it doesn't matter either way.

7 **Now your code will compile.** Update your top-level statements so that they print the TallGuy object's FunnyThingIHave property and then call its Honk method:

```
TallGuy tallGuy = new TallGuy("Jimmy", 76);
tallGuy.TalkAboutYourself();
Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
tallGuy.Honk();
```

Sharpen your pencil

Here's your chance to demonstrate your artistic abilities. On the left, you'll find sets of class and interface declarations. Your job is to draw the associated class diagrams on the right. Don't forget to use a dashed line for implementing an interface and a solid line for inheriting from a class.

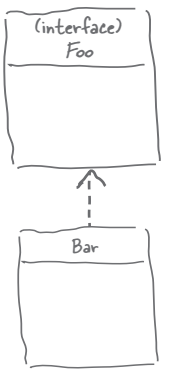
If you're given...

What's the picture?

1

```
interface Foo { }
class Bar : Foo { }
```

We did the first one for you.



2

```
interface Vinn { }
abstract class Vout : Vinn { }
```

3

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

4

```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

You'll need a little more room for #5.

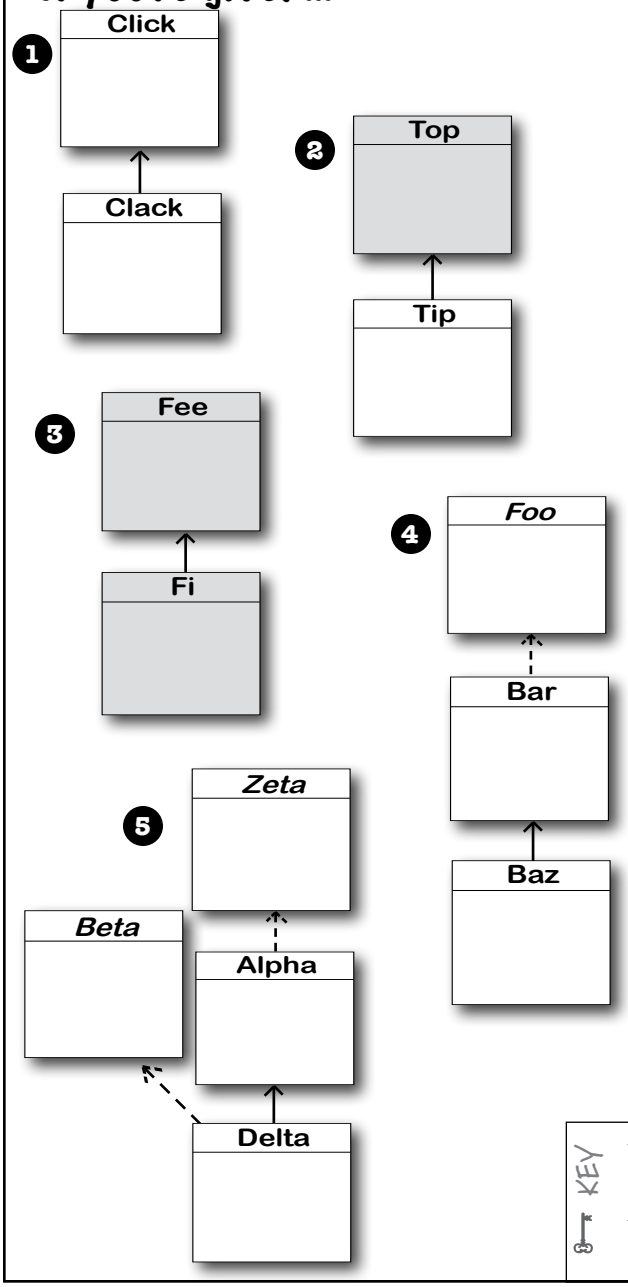
5

```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma,Beta { }
class Delta { }
```

Sharpen your pencil

On the left, you'll find sets of class diagrams. Your job is to turn these into valid C# declarations. **We did number 1 for you.** Notice how the class declarations are just a pair of curly braces **{ }** ? That's because these classes don't have any members. (But they're still valid classes that build!)

If you're given...



What's the declaration?

1 public class Click { }

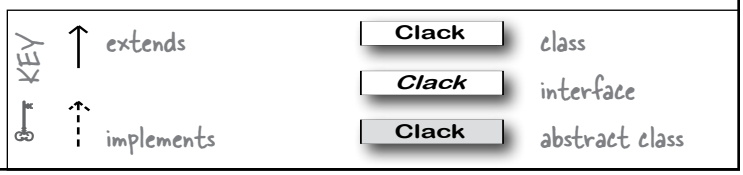
public class Clack : Click { }

2

3

4

5



Fireside Chats

Tonight's talk: **An abstract class and an interface butt heads over the pressing question, "Who's more important?"**

Abstract Class:

I think it's obvious who's more important between the two of us. Programmers need me to get their jobs done. Let's face it. You don't even come close.

You can't really think you're more important than me. You don't even use real inheritance—you only get implemented.

Better? You're nuts. I'm much more flexible than you. Sure, I can't be instantiated—but then, neither can you. Unlike you, I have the **awesome power** of inheritance. The poor saps that extend you can't take advantage of `virtual` and `override` at all!

You might be overstating your power a little bit.

And you think that's a good thing? Ha! When you use me and my subclasses, you know exactly what's going on inside all of us. I can handle any behavior that all of my subclasses need, and they just need to inherit it. Transparency is a powerful thing, kiddo!

Interface:

Nice. This oughta be good.

Great, here we go again. "Interfaces don't use real inheritance." "Interfaces only implement." That's just plain ignorant. Implementation is as good as inheritance. In fact, it's better!

Yeah? What if you want a class that inherits from you ***and*** your buddy? You can't inherit from two classes. You have to choose which class to inherit from. That's just plain rude! There's no limit to the number of interfaces a class can implement. Talk about flexible! With me, a programmer can make a class do anything.

Really, now? Well, let's think about just how powerful I can be for developers that use me. I'm all about the job—when they get an interface reference, they don't need to know anything about what's going on inside that object at all.

9 times out of 10, a programmer wants to make sure an object has certain properties and methods, but doesn't really care how they're implemented.

Abstract Class:

Really? I doubt that—programmers always care what's in their properties and methods.

Yeah, sure, tell a coder they can't code.

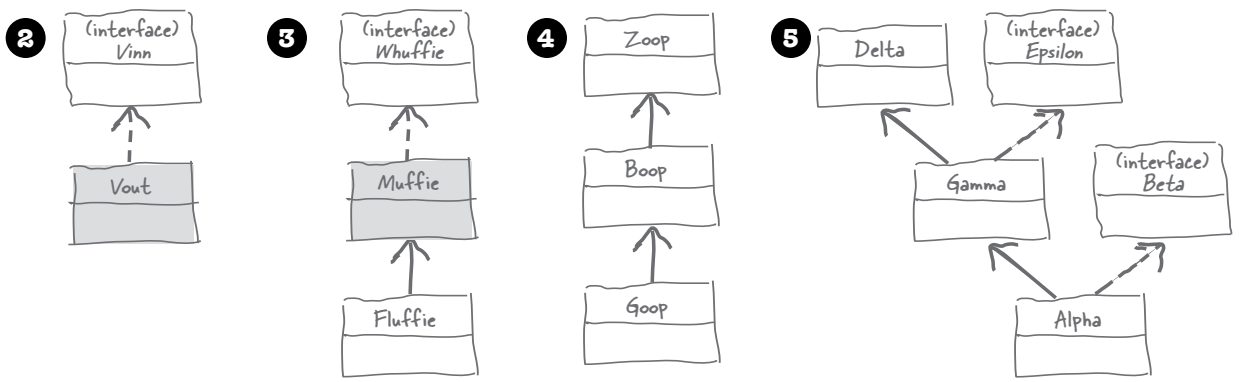
Interface:

OK, sure. Eventually. But think about how many times you've seen a programmer write a method that takes an object that just needs to have a certain method, and it doesn't really matter right at that very moment exactly how the method's built—just that it's there. So bang! The programmer just needs to use an interface. Problem solved!

Ugh, you're so frustrating!

Sharpen your pencil Solution

What's the picture ?



What's the declaration ?

2
```
abstract class Top { }
class Tip : Top { }
```

3
```
abstract class Fee { }
abstract class Fi : Fee { }
```

4
```
interface Foo { }
class Bar : Foo { }
class Baz : Bar { }
```

5
```
interface Zeta { }
class Alpha : Zeta { }
interface Beta { }
class Delta : Alpha, Beta { }
```

Delta inherits from Alpha and implements Beta.

You can't instantiate an interface, but you can reference an interface

Say you need an object that has a Defend method so you can use it in a loop to defend the hive. Any object that implemented the IDefender interface would do. It could be a HiveDefender object, a NectarDefender object, or even a HelpfulLadyBug object. As long as it implements the IDefender interface, that guarantees that it has a Defend method. You just need to call it.

That's where **interface references** come in. You can use one to refer to an object that implements the interface you need and you'll always be sure that it has the right methods for your purpose—even if you don't know much else about it.

If you try to instantiate an interface, your code won't build

You can create an array of IWorker references, but you can't instantiate an interface. What you *can* do is point those references at new instances of classes that implement IWorker. Now you can have an array that holds many different kinds of objects!

If you try to instantiate an interface, the compiler will complain.

```
IDefender barb = new IDefender();
```

THIS WILL *NOT* COMPILE

You can't use the `new` keyword with an interface, which makes sense—the methods and properties don't have any implementation. If you could create an object from an interface, how would it know how to behave?

Use the interface to reference an object you already have

So you can't instantiate an interface…but you ***can* use the interface to make a reference variable**, and use it to reference an object that ***implements*** the interface.

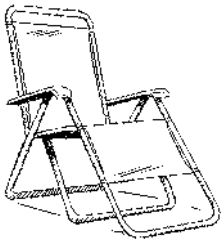
Remember how you could pass a Tiger reference to any method that expects an Animal, because Tiger extends Animal? Well, this is the same thing—you can use an instance of a class that implements IDefender in any method or statement that expects an IDefender.

```
IDefender susan = new HiveDefender();
IDefender ginger = new NectarDefender();
```

Even though this object can do more, when you use an interface reference you only have access to the methods in the interface.

These are ordinary `new` statements, just like you've been using for most of the book. The only difference is that you're **using a variable of type IDefender** to reference them.

We used the interface to declare the "susan" and "ginger" variables, but they're normal references that work just like any other object references.



Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code and output. You may use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a set of classes that will compile and run and produce the output listed.

```
string result = "";
INose[] i = new INose[3];
i[0] = new Acts();
i[1] = new Clowns();
i[2] = new Of2028();
for (int x = 0; x < 3; x++) {
  result += $"{___________} {_________
}\n";
}
Console.WriteLine(result);
Console.ReadKey();

___________ INose {
    ___________;
    string Face { get; }
}

class _________ : _________ {
    public Acts() : base("Acts") { }
    public override ___________  {
        return 5;
    }
}
```

Output

```
5 Acts
7 Clowns
7 Of2028
```

```
abstract class _________ : ________ {
    private string face;
    public virtual string Face {
     _______ { _________ ________ ; }
    }

    public abstract int Ear();

    public Picasso(string face)
    {
        ___________ = face;
    }
}

class _________ : _________ {
    public Clowns() : base("Clowns") { }

    public override int Ear() {
       return 7;
    }
}

class ___________ : _________ {
    public override string Face {
       get { return "Of2028"; }
}
```

Here are the top-level statements.

Note: each snippet from the pool can be used more than once!

Acts( );
INose( );
Of2028( );
Clowns( );
Picasso( );

:
;
class
abstract
interface

i
i( )
i(x)
i[x]

class
5 class
7 class
7 public class

Of2028 [ ] i = new INose[3];
Of2028 [3] i;
INose [ ] i = new INose( );
INose [ ] i = new INose[3];

int Ear()
this
this.
face
this.face

get
set
return

i.Ear(x)
i[x].Ear()
i[x].Face()
i[x].Face

Acts
INose
Of2028
Clowns
Picasso

Pool Puzzle Solution

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code and output. You may use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a set of classes that will compile and run and produce the output listed.

Output

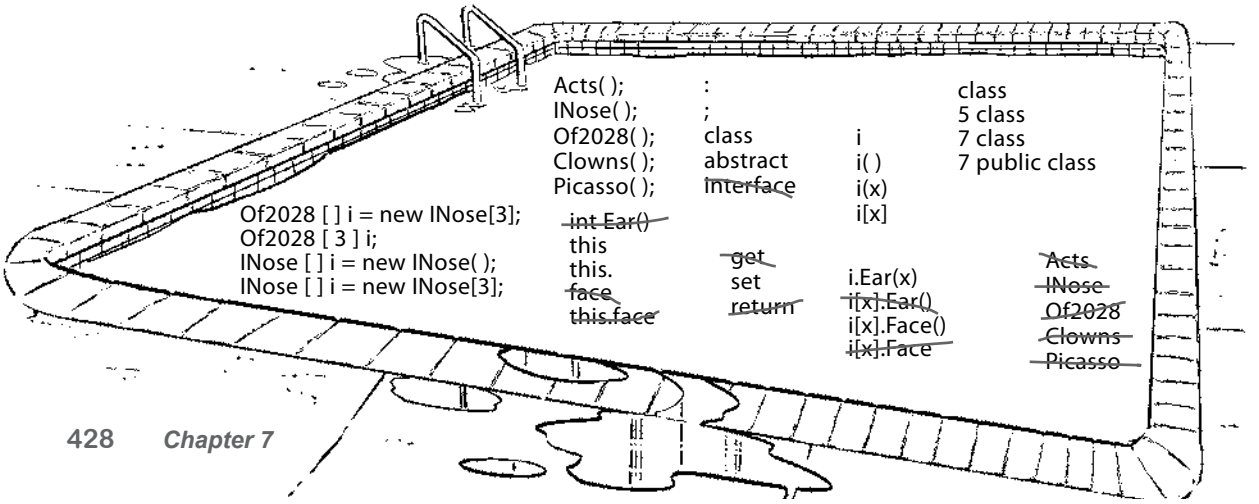
```
5 Acts
7 Clowns
7 Of2028
```

Face is a getter that returns the value of the face property. Both of them are defined in Picasso and inherited into the subclasses.

```
string result = "";
INose[] i = new INose[3];
i[0] = new Acts();
i[1] = new Clowns();
i[2] = new Of2028();
for (int x = 0; x < 3; x++) {
  result += $"{   i[x].Ear()   } {   i[x].Face   }\n";
}
Console.WriteLine(result);
Console.ReadKey();

   interface   INose {
     int Ear()   ;
    string Face { get; }
}

class   Clowns   :   Picasso   {
    public Clowns() : base("Clowns") { }

    public override int Ear() {
       return 7;
    }
}
```

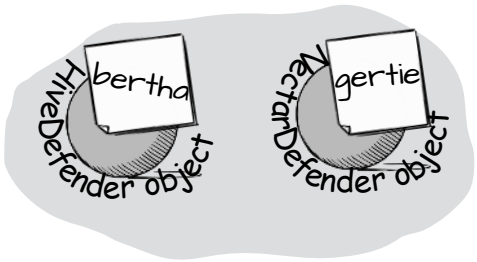
Here's where the Acts class calls the constructor in Picasso, which it inherits from. It passes "Acts" into the constructor, which gets stored in the Face property.

```
abstract class   Picasso   :   INose   {
    private string face;
    public virtual string Face {
      get   {   return   face   ; }
    }

    public abstract int Ear();

    public Picasso(string face)
    {
        this.face   = face;
    }
}

class   Acts   :   Picasso   {
    public Acts() : base("Acts") { }
    public override   int Ear()   {
        return 5;
    }
}

class   Of2028   :   Clowns   {
    public override string Face {
       get { return "Of2028"; } }
   }
}
```



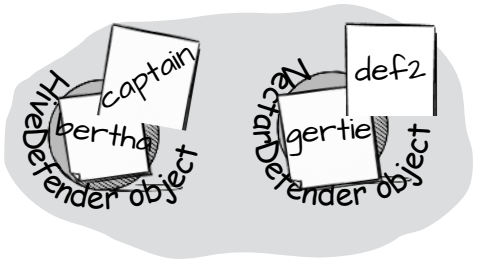
Interface references are ordinary object references

You already know all about how objects live in the heap. When you work with an interface reference, it's just another way to refer to the same objects you've already been using. Let's take a closer look at how interfaces would be used to reference objects in the heap.



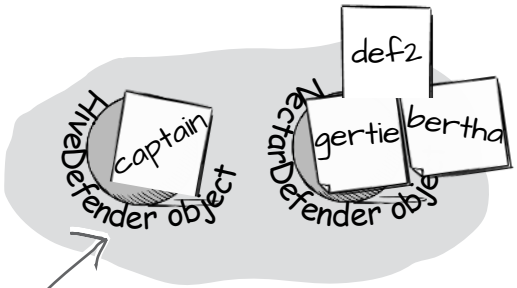
1 **We'll start by creating objects as usual.**
Here's code to create some bees: it creates an instance of HiveDefender and an instance of NectarDefender—and both of those classes implement the IDefender interface.

```
HiveDefender bertha = new HiveDefender();
NectarDefender gertie = new NectarDefender();
```



2 **Next we'll add IDefender references.**
You can use interface references just like you use any other reference type. These two statements use interfaces to create **new references to existing objects**. You can only point an interface reference at an instance of a class that implements it.

```
IDefender def2 = gertie;
IDefender captain = bertha;
```



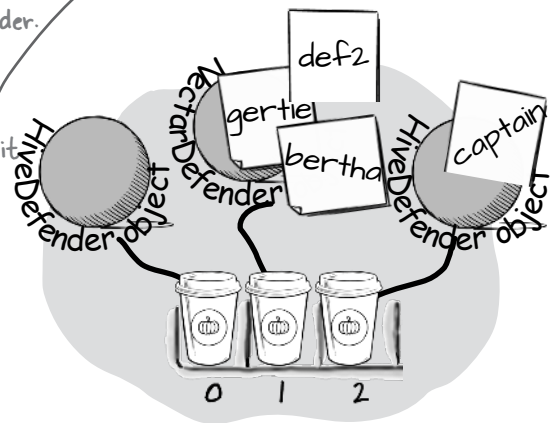
3 **An interface reference will keep an object alive.**
When there aren't any references pointing to an object, it disappears. There's no rule that says those references all have to be the same type! An interface reference is just as good as any other object reference when it comes to keeping track of objects so they don't get garbage-collected.

```
bertha = gertie;
```

← Now bertha points to the NectarDefender.

```
// the captain reference still points to the
// HiveDefender object
```

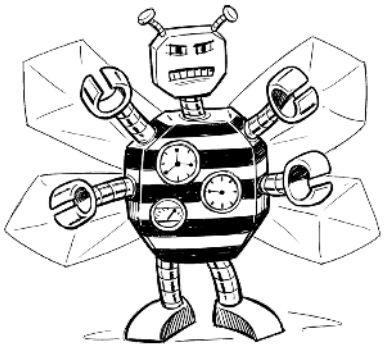
This object didn't disappear from the heap because "captain" still references it.



4 **Use an interface like any other type.**
You can create a new object with a `new` statement and assign it straight to an interface reference variable in a single line of code. You can **use interfaces to create arrays** that can reference any object that implements the interface.

```
IDefender[] defenders = new IDefender[3];
defenders[0] = new HiveDefender();
defenders[1] = bertha;
defenders[2] = captain;
```

The RoboBee 4000 can do a worker bee's job without using valuable honey

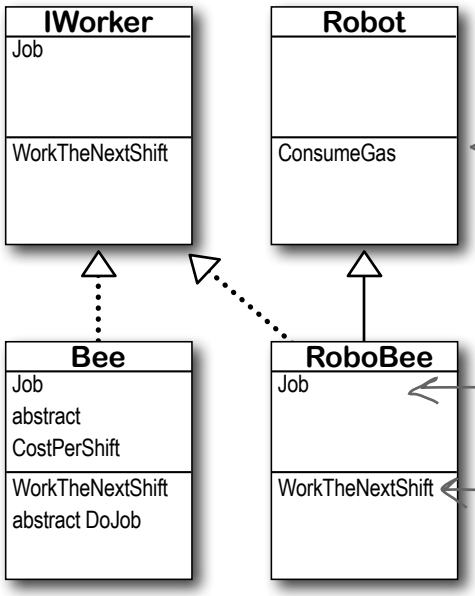


Bee-siness was booming last quarter, and the queen had enough spare budget to buy the latest in hive technology: the RoboBee 4000. It can do the work of three different bees, and best of all it doesn't consume any honey! It's not exactly environmentally friendly, though—it runs on gas. So how can we use interfaces to integrate RoboBee into the hive's day-to-day business?

```
class Robot
{
   public void ConsumeGas() {
     // Not environmentally friendly
   }
}

class RoboBee4000 : Robot, IWorker
{
   public string Job {
     get { return "Egg Care"; }
   }
   public void WorkTheNextShift()
   {
     // Do the work of three bees!
   }
}
```

Let's take a close look at the class diagram to see how we can use an interface to integrate a RoboBee class into the Beehive Management System. Remember, we're using dotted lines to show that an object implements an interface.

We can create an IWorker interface that has the two members that have to do with doing work in the hive.



We'll start with a basic Robot class—everyone knows all robots run on gasoline, so it has a ConsumeGas method.

The Bee class implements the IWorker interface, while the RoboBee class inherits from Robot and implements IWorker. That means it's a robot, but can do the job of worker bees.

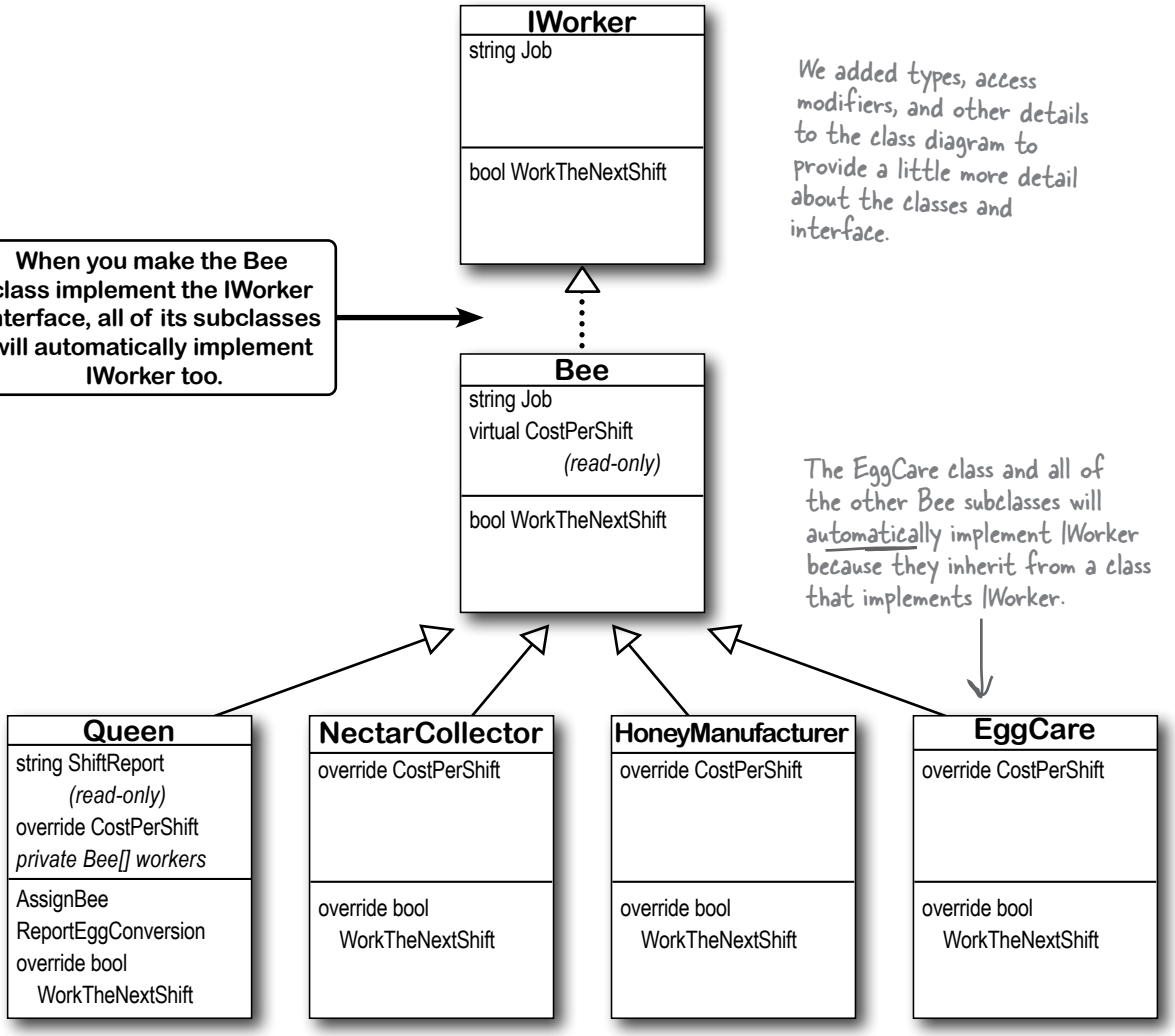
The RoboBee class implements both of the IWorker interface members. We don't have a choice in this—if the RoboBee class doesn't implement everything in the IWorker interface, the code won't compile.

Now all we need to do is modify the Beehive Management System to use the IWorker interface instead of the abstract Bee class any time it needs to reference a worker.

Exercise

Modify the Beehive Management System to create a new IWorker interrace and use it instead of the abstract Bee class any time any code in your app needs to reference a worker.

Your job is to add the IWorker interface to your project, then refactor your code to make the Bee class implement, it and modify the Queen class so it only uses IWorker references. Here's what the updated class diagram looks like:



Here's what you'll need to do:

- Add the IWorker interface to your Beehive Management System project.
- Modify the Bee class to implement IWorker.
- Modify the Queen class to replace any Bee reference with an IWorker reference.

If that doesn't sound like much code, that's because it's not. After you add the interface, you only need to change one line of code in the Bee class and four lines of code in the Queen class.

Exercise Solution

Your job was to modify the Beehive Management System to use the IWorker interface instead of the abstract Bee class any time it needs to reference a worker. You had to add the IWorker interface, then modify the Bee and Queen classes. This didn't take a lot of code—that's because using interfaces doesn't require a lot of extra code.

First you added the IWorker interface to your project

```
interface IWorker
{
    string Job { get; }
    bool WorkTheNextShift();
}
```

Then you modified Bee to implement the IWorker interface

```
abstract class Bee : IWorker
{
    /* The rest of the class stays the same */

}
```

Any class can implement ANY interface as long as it keeps the promise of implementing the interface's methods and properties.

And finally, you modified Queen to use IWorker references instead of Bee references

```
class Queen : Bee
{
    private IWorker[] workers = new IWorker[0];

    private void AddWorker(IWorker worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private string WorkerStatus(string job)
    {
        int count = 0;
        foreach (IWorker worker in workers)
            if (worker.Job == job) count++;
        string s = "s";
        if (count == 1) s = "";
        return $"{count} {job} bee{s}";
    }

   /* Everything else in the Queen class stays the same */
}
```

There's one more Bee reference in the Queen.DoJob method:

```
foreach (IWorker worker in workers)
```

Your code still runs whether or not you change it to IWorker. Why do you think that is?

Try modifying WorkerStatus to change the IWorker in the foreach loop back to a Bee:

```
foreach (IWorker worker in workers)
```

Then run your code—it works just fine! Now try changing it to a NectarCollector. This time you get a System.InvalidCastException.

This happened because every worker in the workers array is a Bee (because every Bee class is a subclass of Bee), but not every worker is a NectarCollector.

there are no Dumb Questions

Q: When I put a property in an interface, it looks just like an automatic property. Does that mean I can only use automatic properties when I implement an interface?

A: No, not at all. It's true that a property inside an interface looks very similar to an automatic property—like the Job property in the IWorker interface on the next page—but they're definitely not automatic properties. You could implement the Job property like this:

```
public Job
{
   get; private set;
}
```

You need that private set, because automatic properties require you to have both a set and a get (even if they're private). Alternatively, you could implement it like this:

```
public Job
{
  get
  {
    return "Egg Care";
  }
}
```

and the compiler will be perfectly happy with that. You can also add a setter—the interface requires a get, but it doesn't say you can't have a set too. (If you use an automatic property to implement it, you can decide for yourself whether you want the set to be private or public.)

Q: Isn't it weird that there are no access modifiers in my interfaces? Should I be marking methods and properties public?

A: You don't need the access modifiers because everything in an interface is automatically public by default. If you have an interface that has this:

```
void Honk();
```

It says that you need a public void method called Honk, but it doesn't say what that method needs to do. It can do anything at all—no matter what it does, the code will compile as long as some method is there with the right signature.

Does that look familiar? That's because we've seen it before—in abstract classes, back in Chapter 6. When you declare methods or properties in an interface without bodies, they're **automatically public and abstract**, just like the abstract members that you used in your abstract classes. They work just like any other abstract methods or properties—because even though you're not using the `abstract` keyword, it's implied. That's why every class that implements an interface **must implement every member**.

The folks who designed C# could have made you mark each of those members public and abstract, but it would have been redundant. So they made the members public and abstract by default to make it all clearer.

Everything in a public interface is automatically public, because you'll use it to define the public methods and properties of any class that implements it.

The IWorker's Job property is a hack

The Beehive Management System uses the Worker.Job property like this: if (worker.Job == job).

Does something seem a bit odd about that? It does to us. We think it's a **hack**, or a clumsy, inelegant solution. Why do we think the Job property is a hack? Imagine what would happen if you had a typo like this:

```
class EggCare : Bee {
   public EggCare(Queen queen) : base("Egg Crae")

   // Oops! Now we've got a bug in the EggCare class,
   // even though the rest of the class is the same.
}
```

We misspelled "Egg Care"—that's a mistake anyone could make! Can you imagine how hard it would be to track down the bugs that this simple typo would cause?

Now the code has no way to figure out if a Worker reference is pointing to an instance of EggCare. That would be a really nasty bug to try to fix. So we know this code is error-prone…but how is it a hack?

We've talked about **separation of concerns**: all of the code to address a specific problem should be kept together. The Job property ***violates the principle of separation of concerns***. If we have a Worker reference, we shouldn't need to check a string to figure out whether it points to an EggCare object or a NectarCollector object. The Job property returns "Egg Care" for an EggCare object and "Nectar Collector" for a NectarCollector object and is only used to check the object's type. But we're already keeping track of that information: ***it's the object's type***.

I think I see where you're going with this. I bet C# gives me a way to figure out the type of an object ***without resorting to a hack****, right?*

That's right! C# gives you tools to work with types.

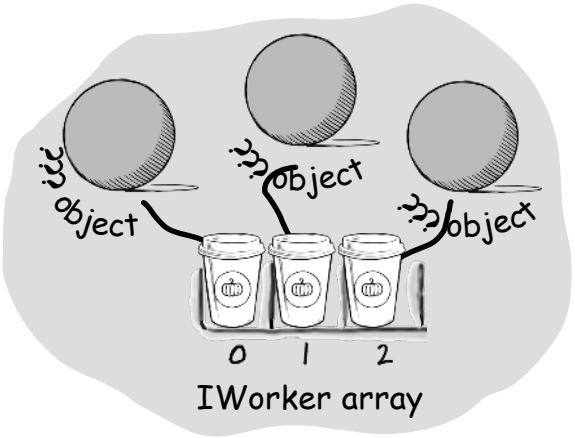
You don't ever need a property like Job to keep track of the type of a class with strings like "Egg Care" or "Nectar Collector." C# gives you tools that let you check the type of an object.

hack, noun.
in engineering, an ugly, clumsy, or inelegant solution to a problem that will be difficult to maintain. *Lila spent an extra hour refactoring the* ***hack*** *in her code so she wouldn't have to deal with bugs later.* (synonym: kludge [klooj])

Use is to check the type of an object

What would it take to get rid of the Job property hack? Right now the queen has her workers array, which means that all she can get is an IWorker reference. She uses the Job property to figure out which workers are EggCare workers and which ones are NectarCollectors:

```
foreach (IWorker worker in workers) {
    if (worker.Job == "Egg Care") {
        WorkNightShift((EggCare)worker);
    }

    void WorkNightShift(EggCare worker) {
       // Code to work the night shift
    }
```



We just saw how that code will fail miserably if we accidentally ***mis***type "Egg C**ra**e" instead of "Egg Care." If you set a HoneyManufacturer's Job to "Egg Care" accidentally, you'll get one of those InvalidCastException errors. It would be great if the compiler could detect problems like that as soon as we write them, just like we use private or abstract members to get it to detect other kinds of problems.

C# gives us a tool to do exactly that: we can use the **`is` keyword** to check an object's type. If you have an object reference, you can **use `is` to find out if it's a specific type**:

```
objectReference is ObjectType newVariable
```

If the object that objectReference is pointing to has ObjectType as its type, then it returns true and creates a new reference called `newVariable` with that type.

So if the queen wants to find all of her EggCare workers and have them work a night shift, she can use the `is` keyword:

```
foreach (IWorker worker in workers) {
    if (worker is EggCare eggCareWorker) {
        WorkNightShift(eggCareWorker);
    }
}
```

The `if` statement in this loop uses `is` to check each IWorker reference. Look closely at the conditional test:

```
worker is EggCare eggCareWorker
```

If the object referenced by the `worker` variable is an EggCare object, that test returns true, and the `is` statement assigns the reference to a new EggCare variable called `eggCareWorker`. This is just like casting, but the `is` statement is **doing the casting for you safely**.

The "is" keyword returns true if an object matches a type, and can declare a variable with a reference to that object.

Use is to access methods in a subclass

Do this!

Let's pull together everything we've talked about so far into a new project by creating a simple class model with Animal at the top, Hippo and Canine classes that extend Animal, and a Wolf class that extends Canine.

Create a new console app and **add these Animal, Hippo, Canine, and Wolf classes** to it:

```
abstract class Animal
{
    public abstract void MakeNoise();
}

class Hippo : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Grunt.");
    }

    public void Swim()
    {
        Console.WriteLine("Splash! I'm going for a swim!");
    }
}

abstract class Canine : Animal
{
    public bool BelongsToPack { get; protected set; } = false;
}

class Wolf : Canine
{
    public Wolf(bool belongsToPack)
    {
        BelongsToPack = belongsToPack;
    }

    public override void MakeNoise()
    {
        if (BelongsToPack)
           Console.WriteLine("I'm in a pack.");
        Console.WriteLine("Arooooo!");
    }

    public void HuntInPack()
    {
        if (BelongsToPack)
            Console.WriteLine("I'm going hunting with my pack!");
        else
            Console.WriteLine("I'm not in a pack.");
    }
}
```

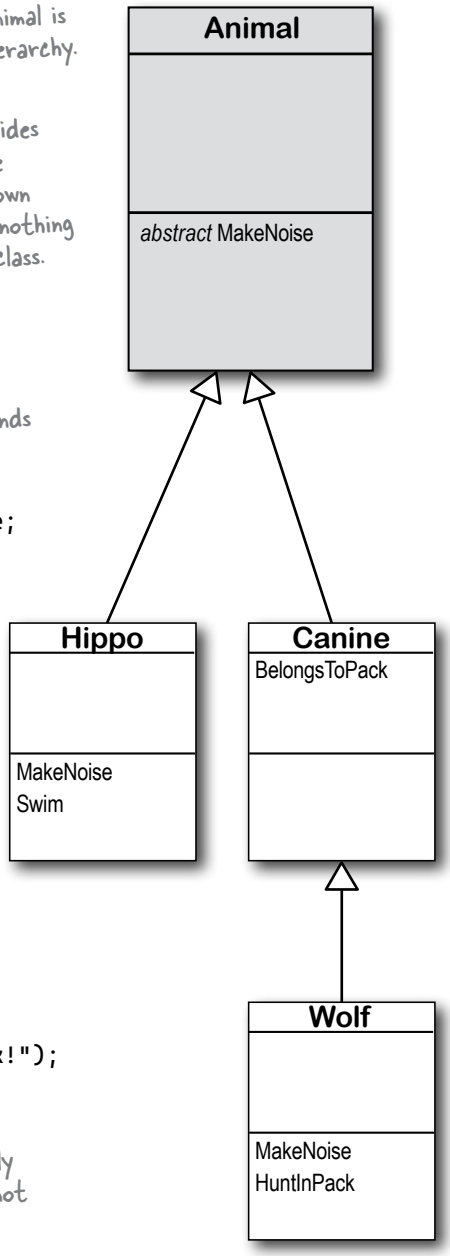
The abstract class Animal is at the top of the hierarchy.

The Hippo subclass overrides the abstract MakeNoise method, and adds its own Swim method that has nothing to do with the Animal class.

The abstract Canine class extends Animal. It has its own abstract property, BelongsToPack.

The Wolf class extends Canine, and adds its own HuntInPack method.

The HuntInPack method is only part of the Wolf class. It's not inherited from a superclass.



Next, fill in the top-level statements for the app. Here's what it does:

- Creates an array of Hippo and Wolf objects, then uses a `foreach` loop to go through each of them.
- Uses the Animal reference to call the MakeNoise method.
- If it's a Hippo, calls its Hippo.Swim method.
- If it's a Wolf, calls its Wolf.HuntInPack method.

In Chapter 6, we learned that we could use different references to call different methods on the same object. When you didn't use the `override` and `virtual` keywords, if your reference variable had the type Locksmith it called Locksmith.ReturnContents, but if it was a JewelThief type it called JewelThief.ReturnContents. We're doing something similar here.

The problem is that if you have an Animal reference pointing to a Hippo object, you can't use it to call Hippo.Swim:

```
Animal animal = new Hippo();
animal.Swim(); // <-- this line will not compile!
```

It doesn't matter that your object is a Hippo. If you're using an Animal variable, you can only access the fields, methods, and properties of Animal. There's a way around this. If you're 100% sure that you have a Hippo object, you can **cast your Animal reference to a Hippo**—then you can access its Hippo.Swim method:

```
Hippo hippo = (Hippo)animal;
hippo.Swim(); // Now you can call the Swim method, but if it's not a Hippo it will
crash
```

The problem is that if it's a Wolf or Canine object, ***the cast will throw an exception because it's not a Hippo***.

The `is` keyword prevents that exception by **safely** checking the type and only doing the cast **if it matches**.

Here are **the top-level statements that use the `is` keyword** to call Hippo.Swim or Wolf.HuntInPack:

```
Animal[] animals =
[
    new Wolf(false),
    new Hippo(),
    new Wolf(true),
    new Wolf(false),
    new Hippo()
];

foreach (Animal animal in animals)
{
    animal.MakeNoise();
    if (animal is Hippo hippo)
    {
        hippo.Swim();
    }

    if (animal is Wolf wolf)
    {
        wolf.HuntInPack();
    }

    Console.WriteLine();
}
```

This foreach loop iterates through the "animals" array. It needs to declare a variable of type Animal to match the array type, but that reference won't let us access Hippo.Swim or Wolf.HuntInPack.

These if statements use the "is" keyword to check if the animal reference is a Hippo or Wolf, and then safely casts it to the hippo or wolf variable so it can call the methods specific to the subclass.

Take a few minutes and use the debugger to really understand what's going on here. Put a breakpoint on the first line of the `foreach` loop; add watches for `animal`, `hippo`, and `wolf`, and step through it.

What if we want different animals to swim or hunt in packs?

Did you know that lions are pack hunters? Or that tigers can swim? And what about dogs, which hunt in packs AND swim? If we want to add the Swim and HuntInPack methods to all of the animals in our zoo simulator model that need them, the `foreach` loop is just going to get longer and longer.

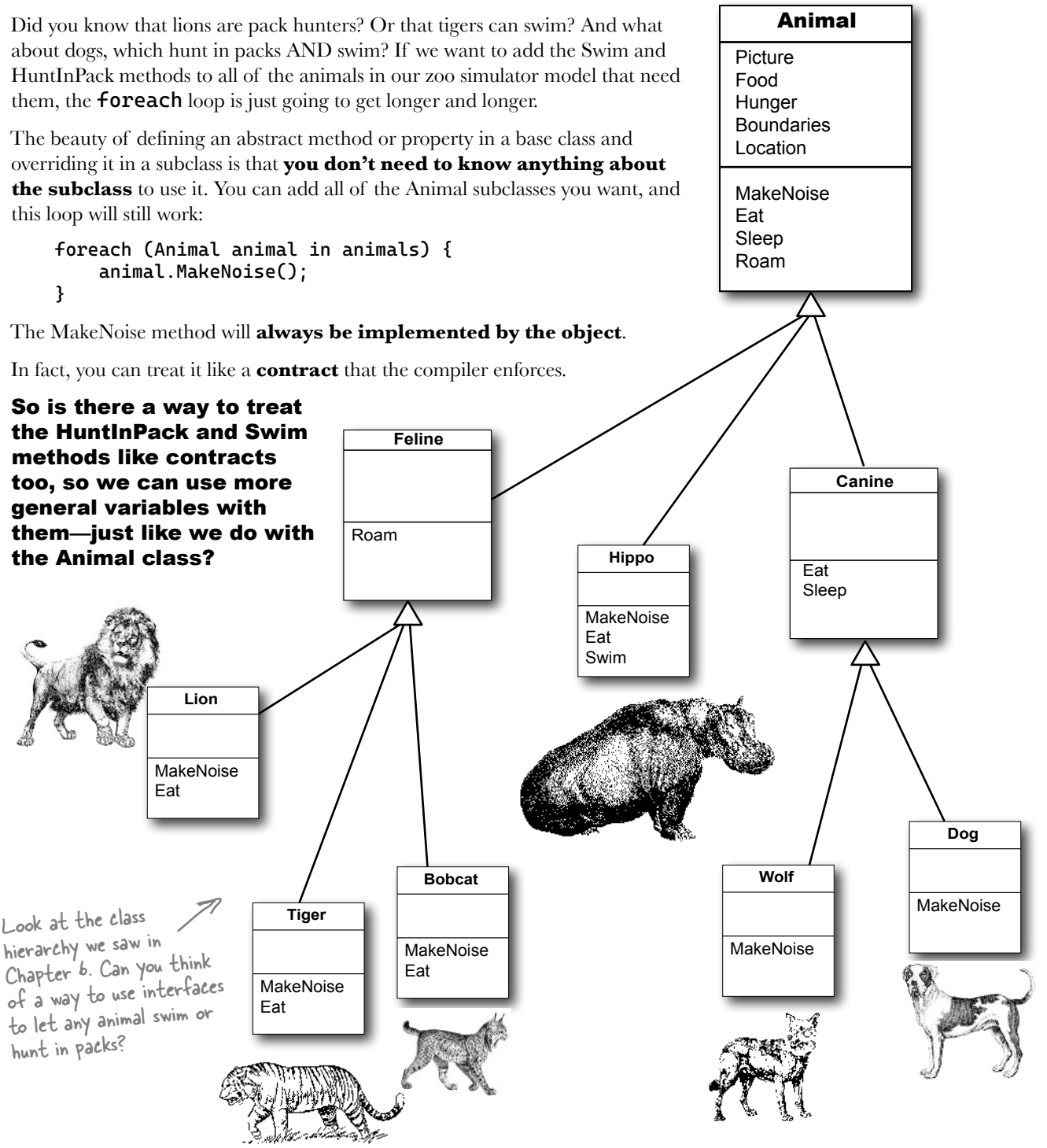
The beauty of defining an abstract method or property in a base class and overriding it in a subclass is that **you don't need to know anything about the subclass** to use it. You can add all of the Animal subclasses you want, and this loop will still work:

```
foreach (Animal animal in animals) {
    animal.MakeNoise();
}
```

The MakeNoise method will **always be implemented by the object**.

In fact, you can treat it like a **contract** that the compiler enforces.

So is there a way to treat the HuntInPack and Swim methods like contracts too, so we can use more general variables with them—just like we do with the Animal class?



Look at the class hierarchy we saw in Chapter 6. Can you think of a way to use interfaces to let any animal swim or hunt in packs?

Use interfaces to work with classes that do the same job

Classes that swim have a Swim method, and classes that hunt in packs have a HuntInPack method. OK, that's a good start. Now we want to write code that works with objects that swim or hunt in packs—and that's where interfaces shine. Let's use the **`interface` keyword** to define two interfaces and **add an abstract member** to each interface:

```
interface ISwimmer
{
    void Swim();
}

interface IPackHunter
{
    void HuntInPack();
}
```

Add this!

Next, **make the Hippo and Wolf classes implement the interfaces** by adding an interface to the end of each class declaration. Use a **colon** (:) to implement an interface, just like you do when you're extending a class. If it's already extending a class, you just add a comma after the superclass and then the interface name. Then you just need to make sure the class **implements all the interface members**, or you'll get a compiler error.

```
class Hippo : Animal, ISwimmer {
    /* The code stays exactly the same - it MUST include the Swim method */
}

class Wolf : Canine, IPackHunter {
    /* The code stays exactly the same - it MUST include the HuntInPack method */
}
```

Use the "is" keyword to check if the Animal is a swimmer or pack hunter

You can use the **`is` keyword** to check if a specific object implements an interface—and it works no matter what other interfaces that object implements or what class it's an instance of. If the `animal` variable references an object that implements the ISwimmer interface, then `animal is ISwimmer` will return true and you can safely cast it to an ISwimmer reference to call its Swim method:

```
foreach (Animal animal in animals)
{
    animal.MakeNoise();
    if (animal is ISwimmer swimmer)
    {
        swimmer.Swim();
    }
    if (animal is IPackHunter hunter)
    {
        hunter.HuntInPack();
    }
    Console.WriteLine();
}
```

What would your code look like if you had 20 different Animal subclasses that swim? You'd need 20 different `if (animal is…)` statements that cast `animal` to each individual subclass to call the Swim method. By using ISwimmer, we only have to check it *once*.

We're using the "is" keyword just like we did earlier, but this time we're using it with interfaces. It still works the same way.

Safely navigate your class hierarchy with is

When you did the exercise to replace Bee with IWorker in the Beehive Management System, were you able to get it to throw the InvalidCastException? ***Here's why it threw the exception.***

You *can* safely convert a HoneyManufacturer reference to an IWorker reference.

All NectarCollectors are Bees (meaning they extend the Bee base class), so you can always use the = operator to take a reference to a NectarCollector and assign it to a Bee variable.

```
HoneyManufacturer lily = new HoneyManufacturer();
Bee hiveMember = lily;
```

And since Bee implements the IWorker interface, you can safely convert it to an IWorker reference too.

```
HoneyManufacturer daisy = new HoneyManufacturer();
IWorker worker = daisy;
```

Those type conversions are safe: they'll never throw an IllegalCastException because they only assign more specific objects to variables with more general types *in the same class hierarchy*.

You *can't* safely convert a Bee reference to a NectarCollector reference.

You can't *safely* go in the other direction—converting a Bee to a NectarCollector—because not all Bee objects are instances of NectarCollector. A HoneyManufacturer is *definitely not* a NectarCollector. So this:

```
IWorker pearl = new HoneyManufacturer();
NectarCollector irene = (NectarCollector)pearl;
```

is an **invalid cast** that tries to cast an object to a variable that doesn't match its type.

The `is` keyword lets you convert types *safely*.

Luckily, **the `is` keyword is safer than casting with parentheses**. It lets you check that the type matches, and only casts the reference to a new variable if the types match.

```
if (pearl is NectarCollector irene) {
    /* Code that uses a NectarCollector object */
}
```

This code will never throw an InvalidCastException because it only executes the code that uses a NectarCollector object if `pearl` is a NectarCollector.

Combining "is" with "if" is safer than casting because it only executes code if the type can be converted.

C# has another tool for safe type conversion: the as keyword

C# gives you another tool for safe casting: the **`as` keyword**. It also does safe type conversion. Here's how it works. Let's say you have an IWorker reference called `pearl`, and you want to safely cast it to a NectarCollector variable `irene`. You can convert it safely to a NectarCollector like this:

```
NectarCollector irene = pearl as NectarCollector;
```

If the types are compatible, this statement sets the `irene` variable to reference the same object as the `pearl` variable. If the type of the object doesn't match the type of the variable, it doesn't throw an exception. Instead, it just **sets the variable to `null`**, which you can check with an `if` statement:

```
if (irene != null) {
   /* Code that uses a NectarCollector object */
}
```

Watch it!

The `is` keyword works differently in very old versions of C#.

The `is` keyword has been in C# for a long time, but it wasn't until C# 7.0 was released in 2017 that `is` let you declare a new variable. So if you're using Visual Studio 2015, you won't be able to do this: `if (pearl is NectarCollector irene) { ... }`

Instead, you'll need to use the `as` keyword to do the conversion, then test the result to see if it's `null`:

```
NectarCollector irene = pearl as NectarCollector;
```

Sharpen your pencil

The array on the left uses types from the Bee class model. Two of these types won't compile—cross them out. On the right are three statements that use the `is` keyword. Write down which values of `i` would make each of them evaluate to `true`.

```
IWorker[] bees = new IWorker[8];
bees[0] = new HiveDefender();
bees[1] = new NectarCollector();
bees[2] = bees[0] as IWorker;
bees[3] = bees[1] as NectarCollector;
bees[4] = IDefender;
bees[5] = bees[0];
bees[6] = bees[0] as Object;
bees[7] = new IWorker();
```

1. `(bees[i] is IDefender)`

   .....

2. `(bees[i] is IWorker)`

   .....

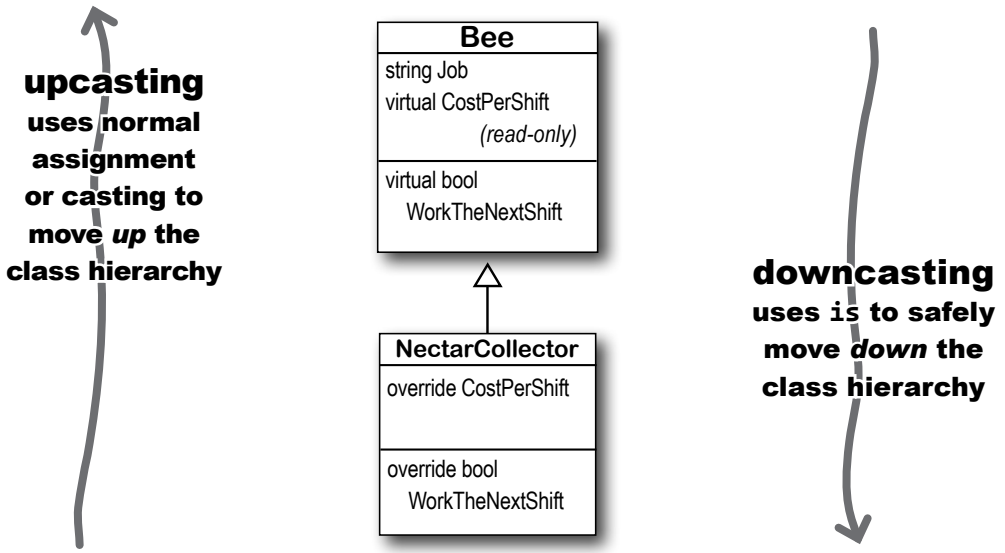
3. `(bees[i] is Bee)`

   .....

Use upcasting and downcasting to move up and down a class hierarchy

Class diagrams typically have the base class at the top, its subclasses below it, their subclasses below them, etc. The higher a class is in the diagram, the more abstract it is; the lower the class is in the diagram, the more concrete it is. "Abstract higher, concrete lower" isn't a hard-and-fast rule—it's a **convention** that makes it easier for us to see at a glance how our class models work.

In Chapter 6, we talked about how you can always use a subclass in place of the base class it inherits from, but you can't always use a base class in place of a subclass that extends it. You can also think about this another way: in a sense, you're **moving up or down the class hierarchy**.



For example, if you start with an instance of NectarCollector:

```
NectarCollector ida = new NectarCollector();
```

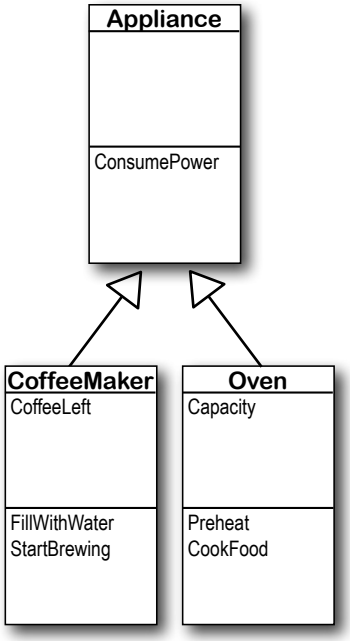
You can use the = operator to do normal assignment (for superclasses) or casting (for interfaces). That's like ***moving up*** the class hierarchy. This is called **upcasting**:

```
// This upcast is safe because every NectarCollector is a Bee
Bee bee = ida;
```

And you can navigate in the other direction by using the `is` operator to safely ***move down*** the class hierarchy. This is called **downcasting**:

```
// You need is to downcast because not every Bee is a NectarCollector
if (bee is NectarCollector rose) {
    rose.WorkTheNextShift();
}
```

A quick example of upcasting



If you're trying to figure out how to cut down your energy bill each month, you don't really care what each of your appliances does—you only care that they consume power. So if you were writing a program to monitor your electricity consumption, you'd probably just write an Appliance class. But if you needed to distinguish a coffee maker from an oven, you'd have to build a class hierarchy and add the methods and properties that are specific to a coffee maker or oven to your CoffeeMaker and Oven classes, which would inherit from an Appliance class that has their common methods and properties.

Then you could write a method to monitor the power consumption:

```
void MonitorPower(Appliance appliance) {
   /* code to add data to a household
      power consumption database */
}
```

If you wanted to use that method to monitor the power consumption for a coffee maker, you could create an instance of CoffeeMaker and pass its reference directly to the method:

```
CoffeeMaker misterCoffee = new CoffeeMaker();
MonitorPower(misterCoffee);
```

This is a great example of upcasting. Even though the MonitorPower method takes a reference to an Appliance object, you can pass it the misterCoffee reference because CoffeeMaker is a subclass of Appliance.

Sharpen your pencil Solution

The array on the left uses types from the Bee class model. Two of these types won't compile—cross them out. On the right are three statements that use the `is` keyword. Write down which values of `i` would make each of them evaluate to `true`.

```
IWorker[] bees = new IWorker[8];
bees[0] = new HiveDefender();
bees[1] = new NectarCollector();
bees[2] = bees[0] as IWorker;
bees[3] = bees[1] as NectarCollector;
~~bees[4] = IDefender;~~
bees[5] = bees[0];
bees[6] = bees[0] as Object;
~~bees[7] = new IWorker();~~
```

This line casts the IWorker to a NectarCollector but then stores it as an IWorker reference again.

1. `(bees[i] is IDefender)`

   0, 2, 5, and 6

   Elements 0, 2, 5, and 6 in the array all point to the same HiveDefender object.

2. `(bees[i] is IWorker)`

   0, 1, 2, 3, 5, 6

3. `(bees[i] is Bee)`

   0, 1, 2, 3, 5, 6

   All of the objects extend Bee, and Bee implements IWorker, so they're all Bees and IWorkers.

Upcasting turns your CoffeeMaker into an Appliance

When you substitute a subclass for a base class—like substituting a CoffeeMaker for an Appliance, or a Hippo for an Animal—it's called **upcasting**. It's a really powerful tool to use when you build class hierarchies. The only drawback to upcasting is that you can only use the properties and methods of the base class. In other words, when you treat a CoffeeMaker like an Appliance, you can't tell it to make coffee or fill it with water. You can tell whether or not it's plugged in, since that's something you can do with any Appliance (which is why the PluggedIn property is part of the Appliance class).

You don't need to add this code to an app—just read through the code and start to get a sense of how upcasting and downcasting work. You'll get *lots* of practice with them later in the book.

1 **Let's create some objects.**

Let's start by creating instances of the CoffeeMaker and Oven classes as usual:

```
CoffeeMaker misterCoffee = new CoffeeMaker();
Oven oldToasty = new Oven();
```

2 **What if we want to create an array of Appliances?**

You can't put a CoffeeMaker in an Oven[] array, and you can't put an Oven in a CoffeeMaker[] array. You ***can*** put both of them in an Appliance[] array:

```
Appliance[] kitchenWare = new Appliance[2];
kitchenWare[0] = misterCoffee;
kitchenWare[1] = oldToasty;
```

You can use upcasting to create an array of Appliances that can hold both CoffeeMakers and Ovens.

3 **But you can't treat just any Appliance like an Oven.**

When you've got an Appliance reference, you can only access the methods and properties that have to do with appliances. You **can't** use the CoffeeMaker methods and properties through the Appliance reference ***even if you know it's really a CoffeeMaker***. So these statements will work just fine, because they treat a CoffeeMaker object like an Appliance:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

But as soon as you try to use it like a CoffeeMaker:

```
powerConsumer.StartBrewing();
```

This line won't compile because powerConsumer is an Appliance reference, so it can only be used to do Appliance things.

your code won't compile, and the IDE will display an error:

CS1061	'Appliance' does not contain a definition for 'StartBrewing' and no accessible extension method 'StartBrewing' accepting a first argument of type 'Appliance' could be found (are you missing a using directive or an assembly reference?)

Once you upcast from a subclass to a base class, you can only access the methods and properties that **match the reference** that you're using to access the object.

powerConsumer is an Appliance reference pointing to a CoffeeMaker object.

Downcasting turns your Appliance back into a CoffeeMaker

Upcasting is a great tool, because it lets you use a CoffeeMaker or an Oven anywhere you just need an Appliance. But it's got a big drawback—if you're using an Appliance reference that points to a CoffeeMaker object, you can only use the methods and properties that belong to Appliance. That's where **downcasting** comes in: that's how you take your **previously upcast reference** and change it back. You can figure out if your Appliance is really a CoffeeMaker using the **`is`** keyword, and if it is you can convert it back to a CoffeeMaker.

1. **We'll start with the CoffeeMaker we already upcast.**
Here's the code that we used:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

2. **What if we want to turn the Appliance back into a CoffeeMaker?**
Let's say we're building an app that looks in an array of Appliance references so it can make our CoffeeMaker start brewing. We can't just use our Appliance reference to call the CoffeeMaker method:

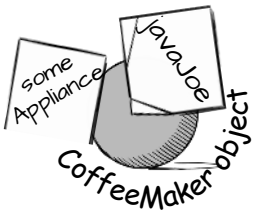
```
Appliance someAppliance = appliances[5];
someAppliance.StartBrewing()
```

That statement won't compile—you'll get that "'Appliance' does not contain a definition for 'StartBrewing'" compiler error because StartBrewing is a member of CoffeeMaker but you're using an Appliance reference.

Here's our Appliance reference that points to a CoffeeMaker object. You can only use it to access members of the Appliance class.

3. **But since we know it's a CoffeeMaker, let's use it like one.**
The **`is`** keyword is the first step. Once you know that you've got an Appliance reference that's pointing to a CoffeeMaker object, you can use **`is`** to downcast it. That lets you use the CoffeeMaker class's methods and properties. Since CoffeeMaker inherits from Appliance, it still has its Appliance methods and properties.

```
if (someAppliance is CoffeeMaker javaJoe) {
   javaJoe.StartBrewing();
}
```

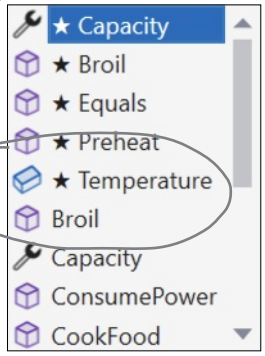


The javaJoe reference points to the same CoffeeMaker object as powerConsumer, but it's a CoffeeMaker reference, so it can call the StartBrewing method.

Upcasting and downcasting work with interfaces too

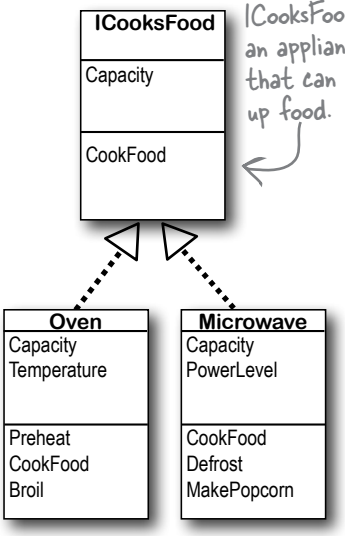
Interfaces work really well with upcasting and downcasting. Let's add an ICooksFood interface for any class that can heat food. Next, we'll add a Microwave class—both Microwave and Oven implement the ICooksFood interface. Now a reference to an Oven object can be an ICooksFood reference or an Oven reference. That means we have three different types of references that could point to an Oven object—and each of them can access different members, depending on the reference's type. Luckily, the IDE's IntelliSense can help you figure out exactly what you can and can't do with each of them:

```
Oven misterToasty = new Oven();
misterToasty.
```



When you have an Oven reference, you can access the Oven members, including Temperature, Preheat, and Broil.

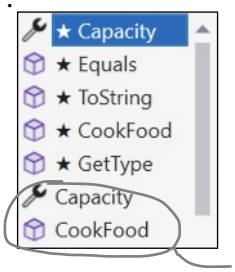
As soon as you type the dot, the IntelliSense window will pop up with a list of all of the members you can use. `misterToasty` is an Oven reference pointing to an Oven object, so it can access *all* of the methods and properties. It's the *most specific type*, so you can only point it at Oven objects.



Any class that implements ICooksFood is an appliance that can heat up food.

To access ICooksFood interface members, convert it to an ICooksFood reference:

```
if (misterToasty is ICooksFood cooker) {
cooker.
```



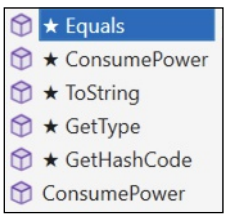
`cooker` is an ICooksFood reference pointing to that same Oven object. It can only access ICooksFood members, but it can also point to a Microwave object.

An ICooksFood reference can only access members that are part of the interface, Capacity and CookFood.

Three different references that point to the same object can access different methods and properties, depending on the reference's type.

This is the same Oven class that we used earlier, so it also extends the Appliance base class. If you use an Appliance reference to access the object, you'll only see members of the Appliance class:

```
if (misterToasty is Appliance powerConsumer)
    powerConsumer.
```



Appliance only has one member, ConsumePower, so that's all you'll see in the dropdown.

`powerConsumer` is an Appliance reference. It only lets you get to the public fields, methods, and properties in Appliance. It's more general than an Oven reference (so you could point it at a CoffeeMaker object if you wanted to).

there are no Dumb Questions

Q: So back up—I think you told me that I can always upcast but I can't always downcast. Why?

A: Because an upcast won't work if you're trying to set an object equal to a class that it doesn't inherit from or an interface that it doesn't implement. The compiler can figure out immediately that you didn't upcast properly and give you an error. When we say "You can always upcast but you can't always downcast," it's just like saying "Every oven is an appliance but not every appliance is an oven."

Q: I'm still having trouble wrapping my head around interfaces. Is there another way to think about them?

A: A lot of folks like to say that an interface is like a **contract**. In fact, "How is an interface like a contract?" is a really common question during job interviews. When you make your class implement an interface, you're telling the compiler that you promise to put certain methods into it. The compiler will hold you to that promise. That's like a court forcing you to stick to the terms of a contract. On the other end of the contract, when you have a reference to an instance of a class that implements that interface, you're guaranteed that it will have all of the interface members.

If that helps you understand interfaces, then definitely think of them that way. It's certainly a great way to think about them!

Q: You've talked about interfaces as a way to talk about a job that a class does, and now as a contract. Those seem helpful, but is there another way of thinking about it?

A: Sure. For some folks, it's easier to learn how interfaces work if we think of an interface as a kind of **checklist**. The compiler runs through the checklist to make sure that you actually put all of the methods from the interface into your class. If you didn't, it'll give you an error out and not let you compile.

Q: Why would I want to use an interface? It seems like it's just adding restrictions, without actually changing my class at all.

A: Because when your class implements an interface, you can use that interface as a type to declare a reference that can point to any instance of a class that implements it. That's really useful to you—it lets you create one reference type that can work with a whole bunch of different kinds of objects.

Here's a quick example. A horse, an ox, a mule, and a steer can all pull a cart. In our zoo simulator, Horse, Ox, Mule, and Steer would all be different classes. Let's say you had a cart-pulling ride in your zoo, and you wanted to create an array of any animal that could pull carts around. Uh-oh—you can't just create an array that will hold all of those. If they all inherited from the same base class you could create an array of those, but it turns out that they don't. So what'll you do?

That's where interfaces come in handy. You can create an IPuller interface that has methods for pulling carts around. Then you can declare your array like this:

```
IPuller[] pullerArray;
```

Now you can put a reference to any animal you want in that array, as long as it implements the IPuller interface.

An interface is like a contract that guarantees an object has specific methods and properties. An interface is also like a checklist that the compiler runs through to make sure your class implemented them. Those are both useful ways to think about interfaces.

Interfaces can inherit from other interfaces

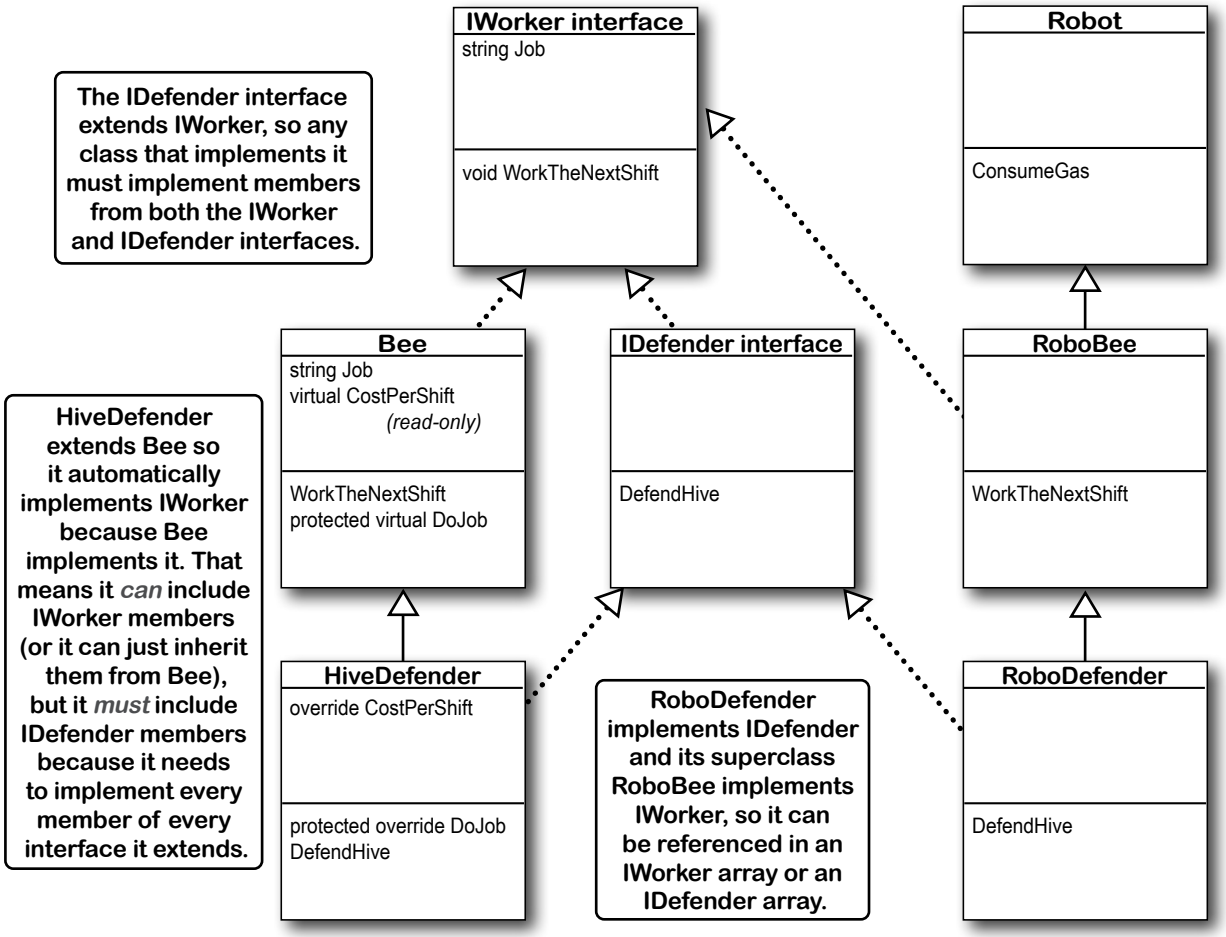
As we've mentioned, when one class inherits from another, it gets all of the methods and properties from the base class. **Interface inheritance** is simpler. Since there's no actual method body in any interface, you don't have to worry about calling base class constructors or methods. The inherited interfaces **accumulate all of the members** of the interfaces that they extend.

So what does this look like in code? Let's add an IDefender interface that inherits from IWorker:

```
interface IDefender : IWorker {
    void DefendHive();
}
```

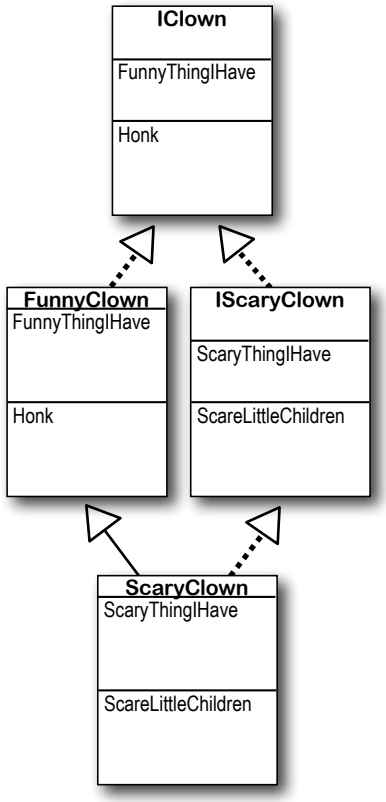
Use a colon (:) to make an interface *extend* another interface.

When a class implements an interface, it must implement every property and method in that interface. If that interface inherits from another one, then all of *those* properties and methods need to be implemented too. So any class that implements IDefender not only must implement all of the IDefender members, but also all of the IWorker members. Here's a class model that includes IWorker and IDefender, and **two separate hierarchies** that implement them.



Exercise

Create a new console app with classes that implement the IClown interface. Can you figure out how to get the code at the bottom to build?



1. Start with the IClown interface you created earlier:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

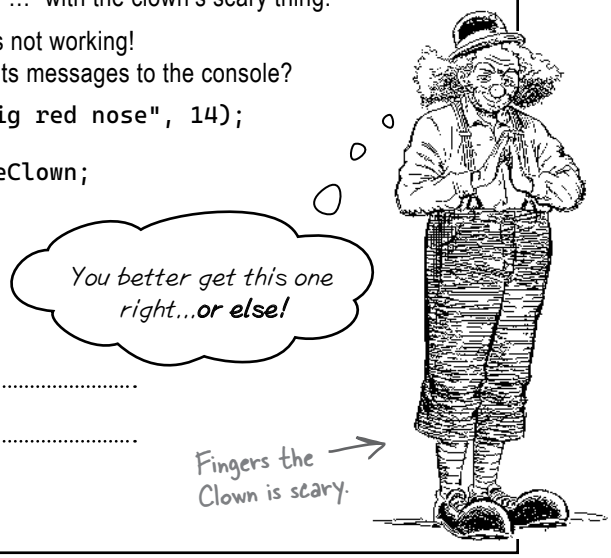
2. Extend IClown by creating **a new interface called IScaryClown** that extends IClown. It should have a string property called ScaryThingIHave with a get accessor but no set accessor, and a void method called ScareLittleChildren.

3. Create these classes that implement the interfaces:

   - A **class called FunnyClown** that implements IClown. It uses a private string variable called funnyThingIHave to store a funny thing. The FunnyThingIHave getter uses funnyThingIHave as a backing field. Use a constructor that takes a parameter and uses it to set the private field. The Honk method prints: `"Hi kids! I have "` followed by the funny thing and a period.
   - A **class called ScaryClown** that implements IScaryClown and extends FunnyClown. It uses a private variable to store an integer called scaryThingCount. The constructor sets both the scaryThingCount field and the funnyThingIHave field that ScaryClown inherited from FunnyClown. The ScaryThingIHave getter returns a string with the number from the constructor followed by *spiders*. The ScareLittleChildren method writes *"Boo! Gotcha! Look at my…!"* to the console, replacing "…" with the clown's scary thing.

4. Here's the code for the top-level statements—but something's not working! Can you **figure out how to fix this code** so it builds and prints messages to the console?

```
IClown fingersTheClown = new ScaryClown("a big red nose", 14);
fingersTheClown.Honk();
IScaryClown iScaryClownReference = fingersTheClown;
iScaryClownReference.ScareLittleChildren();
```



Before you run the code, **write down the output** that app will print to the console (once you fix it):

.....

.....

Then run the code and see if you got the answer right.

Exercise Solution

The IScaryClown interface *inherits* from the IClown interface. That means any class that implements IScaryClown not only needs to have a ScaryThingIHave property and a ScareLittleChildren method, but also a FunnyThingIHave property and a Honk method.

The IScaryClown interface extends IClown and adds a property and a method:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

The FunnyClown class implements the IClown interface and uses a constructor to set a backing field:

```
class FunnyClown : IClown
{
    private string funnyThingIHave;

    public string FunnyThingIHave { get { return funnyThingIHave; } }

    public FunnyClown(string funnyThingIHave)
    {
        this.funnyThingIHave = funnyThingIHave;
    }

    public void Honk()
    {
        Console.WriteLine($"Hi kids! I have {funnyThingIHave}.");
    }
}
```

These are just like the constructors and backing fields we used in Chapter 5.

The ScaryClown class extends the FunnyClown class and implements the IScaryClown interface. Its constructor uses the `base` keyword to call the FunnyClown constructor to set the private backing field:

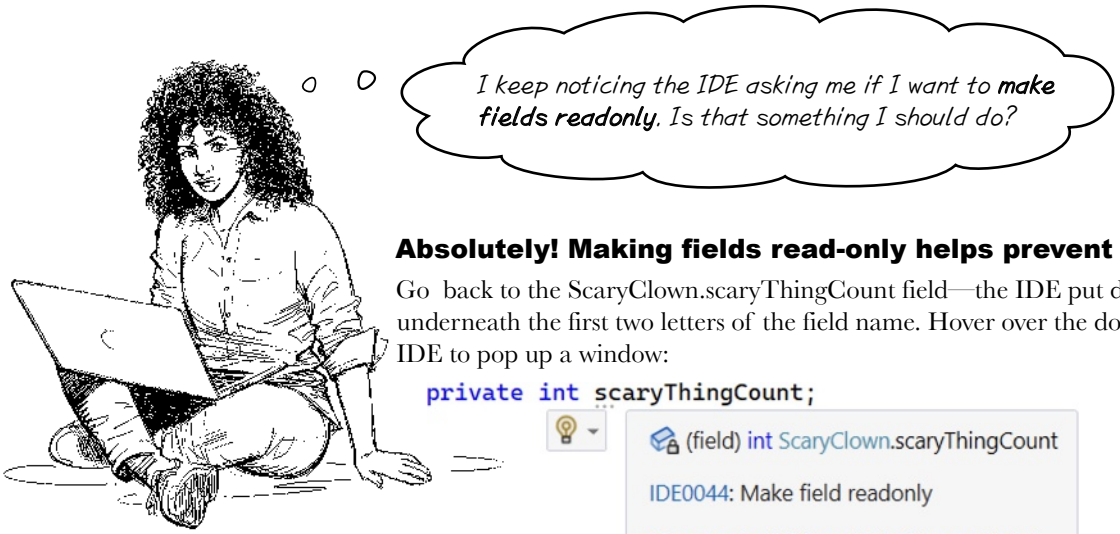
```
class ScaryClown : FunnyClown, IScaryClown
{

    private int scaryThingCount;

    public ScaryClown(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }

    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }

    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}!");
    }
}
```

FunnyClown.funnyThingIHave is a private field, so ScaryClown can't access it—it needs to use the `base` keyword to call the FunnyClown constructor.

To fix the top-level statements, replace the last two lines with this code that uses the **`is`** keyword:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareLittleChildren();
}
```

You can set a FunnyClown reference equal to a ScaryClown object because ScaryClown inherits from FunnyClown. You can't set any IScaryClown reference to just any clown, because you don't know if that clown is scary. That's why you need to use the `is` keyword.



Absolutely! Making fields read-only helps prevent bugs.

Go back to the ScaryClown.scaryThingCount field—the IDE put dots underneath the first two letters of the field name. Hover over the dots to get the IDE to pop up a window:

```
private int scaryThingCount;
```

(field) int ScaryClown.scaryThingCount

IDE0044: Make field readonly

Show potential fixes (Alt+Enter or Ctrl+.)

Press Ctrl-period (Windows) or ⌘-period (macOS). to pop up a list of actions, and choose **"Add readonly modifier"** to add the **`readonly` keyword** to the declaration:

```
private readonly int scaryThingCount;
```

Now the field can only be set when it's declared or in the constructor. If you try to change its value anywhere else in the method, you'll get a compiler error:

CS0191	A readonly field cannot be assigned to (except in a constructor or a variable initializer)

The `readonly` keyword…just another way C# helps you keep your data safe.

The "readonly" keyword

An important reason that we use encapsulation is to prevent one class from accidentally overwriting another class's data. What's preventing a class from overwriting its own data? The "readonly" keyword can help with that. Any field that you mark readonly can only be modified in its declaration or in the constructor.

there are no Dumb Questions

Q: Why would I want to use an interface instead of just writing all of the methods I need directly into my class?

A: When you use interfaces, you still write methods in your classes. Interfaces let you group those classes by the kind of work they do. They help you be sure that every class that's going to do a certain kind of work does it using the same methods. The class can do the work however it needs to, and because of the interface, you don't need to worry about how it gets the job done.

Here's an example: you can have Truck and Sailboat classes that both implement ICarryPassenger. Say the ICarryPassenger interface stipulates that any class that implements it has to have a ConsumeEnergy method. Your program could use them both to carry passengers even though the Sailboat class's ConsumeEnergy method uses wind power and the Truck class's method uses diesel fuel.

Imagine if you didn't have the ICarryPassenger interface. Then it would be tough to tell your program which vehicles could carry people and which couldn't. You would have to look through each class that your program might use and figure out whether or not there was a method for carrying people from one place to another. Then you'd have to call each of the vehicles your program was going to use with whatever method was defined for carrying passengers. And since there's no standard interface, they could be named all sorts of things or buried inside other methods. You can see how that gets confusing pretty fast.

Q: Why do I need to use properties in interfaces? Can't I just include fields?

A: Good question. An interface only defines the way a class should do a specific kind of job. It's not an object by itself, so you can't instantiate it and it can't store information. If you added a field that was just a variable declaration, then C# would have to store that data somewhere—and an interface can't store data by itself. A property is a way to make something that looks like a field to other objects, but since it's really a method, it doesn't actually store any data.

Q: What's the difference between a regular object reference and an interface reference?

A: You already know how a regular, everyday object reference works. If you create an instance of Skateboard called `vertBoard`, and then a new reference to it called `halfPipeBoard`, they both point to the same thing. But if Skateboard implements the interface IStreetTricks and you create an interface reference to Skateboard called `streetBoard`, it will only know the methods in the Skateboard class that are also in the IStreetTricks interface.

All three references are actually pointing to the same object. If you call the object using the `halfPipeBoard` or `vertBoard` references, you'll be able to access any method or property in the object. If you call it using the `streetBoard` reference, you'll only have access to the methods and properties in the interface.

Q: Then why would I ever want to use an interface reference, if it limits what I can do with the object?

A: Interface references give you a way of working with a bunch of different kinds of objects that do the same thing. You can create an array using the interface reference type that will let you pass information to and from the methods in ICarryPassenger whether you're working with a Truck object, a Horse object, a Unicycle object, or a Car object. The way each of those objects does the job is probably a little different, but with interface references, you know that they all have the same methods that take the same parameters and have the same return types. So, you can call them and pass information to them in exactly the same way.

Q: Remind me again why would I make a class member protected instead of private or public?

A: Because it helps you encapsulate your classes better. There are a lot of times when a subclass needs access to some internal part of its base class. For example, if you need to override a property, it's pretty common to use the backing field in the base class in the get accessor, so that it returns some sort of variation of it. When you build classes, you should only make something public if you have a reason to do it. Using the `protected` access modifier lets you expose it only to the subclass that needs it, and keep it private from everyone else.

Interface references only know about the methods and properties that are defined in the interface.

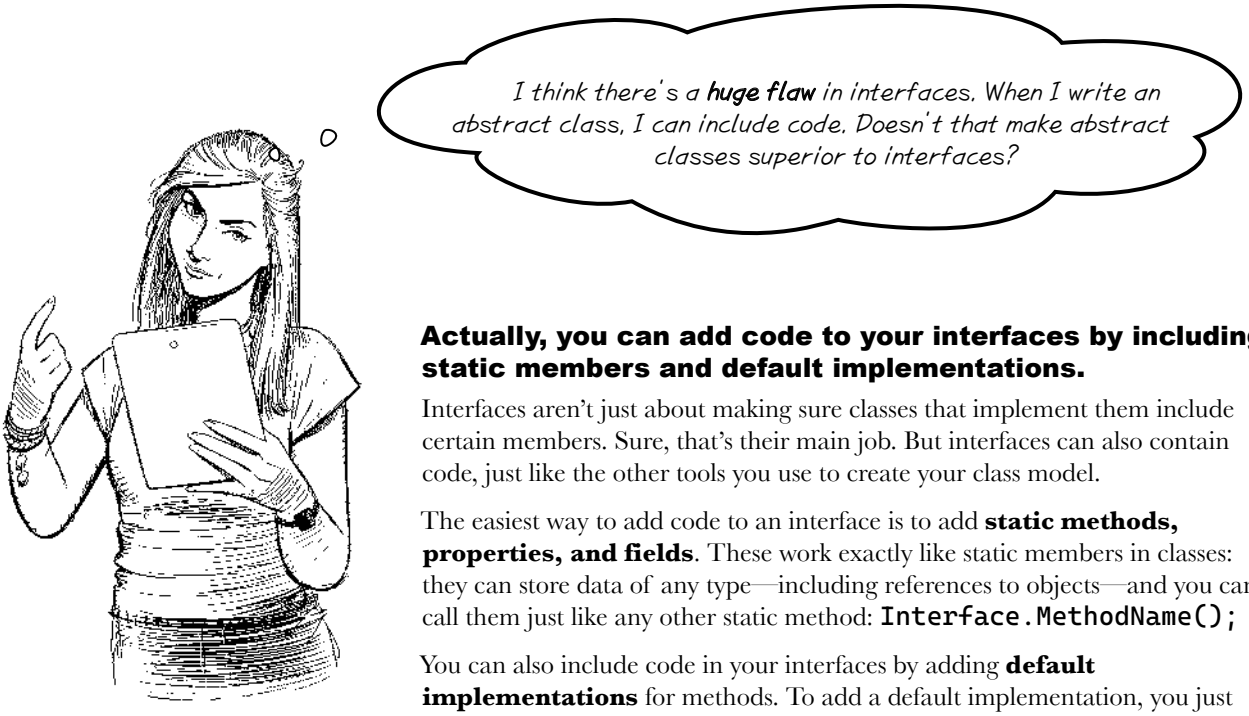
Bullet Points

- An **interface** defines methods and properties that a class must implement.
- Interfaces define their **required members** using abstract methods and properties.
- By default, all interface members are **public and abstract** (so the `public` and `abstract` keywords are typically left off each member).
- When you use a **colon (:)** to make a class implement an interface, the class **must implement *all* of its members** or the code won't compile.
- A class can **implement multiple interfaces** (and it doesn't run into the Deadly Diamond of Death because the interfaces have no implementation).
- Interfaces are really useful because they let **unrelated** classes do the **same job**.
- Whenever you create an interface, you should make its name start with an **uppercase `I`** (that's just a convention; it's not enforced by the compiler).
- We use dashed arrows to draw interface implementation relationships in our class diagrams.
- You **can't use the `new` keyword** to instantiate an interface because its members are abstract.
- You can use an **interface as a type** to reference an object that implements it.
- Any class can **implement *any* interface** as long as it keeps the promise of implementing the interface's methods and properties.
- Everything in a public interface is **automatically public**, because it defines public methods and properties of any class that implements it.
- A **hack** is an ugly, clumsy, or inelegant solution to a problem that will be difficult to maintain.
- The **`is` keyword** returns true if an object matches a type. You can also use it to declare a variable and set it to reference the object you're checking.
- **Upcasting** typically means using normal assignment or casting to move up the class hierarchy, or assign a superclass variable to reference a subclass object.
- The `is` keyword lets you **downcast**—safely move down the class hierarchy—to use a subclass variable to reference a superclass object.
- Upcasting and downcasting **work with interfaces** too—you can upcast an object reference to an interface reference, or downcast from an interface reference.
- The **`as` keyword** is like a cast, except instead of throwing an exception it returns null if the cast is invalid.
- When you mark a field with the **`readonly` keyword,** it can only be set in the field initializer or constructor.

Make it Stick

To remember how interfaces work: you extend a class, but implement an interface. Extending something means taking what's already there and stretching it out (in this case, by adding behavior). Implementing means putting an agreement into effect—you've agreed to add all the interface members (and the compiler holds you to that agreement).

Look up "implement" in the dictionary—one definition is "to put a decision, plan, or agreement into effect."



Actually, you can add code to your interfaces by including static members and default implementations.

Interfaces aren't just about making sure classes that implement them include certain members. Sure, that's their main job. But interfaces can also contain code, just like the other tools you use to create your class model.

The easiest way to add code to an interface is to add **static methods, properties, and fields**. These work exactly like static members in classes: they can store data of any type—including references to objects—and you can call them just like any other static method: `Interface.MethodName();`

You can also include code in your interfaces by adding **default implementations** for methods. To add a default implementation, you just add a method body to the method in your interface. This method is not part of the object—this is not the same as inheritance—and you can only access it using an interface reference. It can call methods implemented by the object, as long as they're part of the interface.

Watch it!

Default interface implementations are a new(-ish) C# feature.

If you're using an old version of Visual Studio, you may not be able to use default implementations because they were added in C# 8.0, which first shipped in Visual Studio 2019 version 16.3.0, released in September 2019. Support for the current version of C# may not be available in older versions of Visual Studio.

Here's a really useful page on the Microsoft website that can help you sort out these versions:

https://docs.microsoft.com/dotnet/csharp/language-reference/configure-language-version

Interfaces can have static members

Everybody loves it when way too many clowns pack themselves into a tiny clown car! So let's update the IClown interface to add static methods that generate a clown car description. Here's what we'll add:

- A clown car is only funny if it's packed with clowns, so we'll add a static int property with a static backing field and a setter that only accepts values over 10.
- We'll add the **protected** access modifier to the static backing field—that way any interface that extends IClown can access it, but the rest of the code can't modify it.
- A method called ClownCarDescription returns a string that describes the clown car.

IClown
FunnyThingIHave protected static CarCapacity
Honk static ClownCarDescription

Here's the code—it uses a static field, property, and method just like you'd see in a class:

```
interface IClown
{
   string FunnyThingIHave { get; }
   void Honk();

   private static int carCapacity = 12;

   public static int CarCapacity {
      get { return carCapacity; }
      set {
         if (value > 10) carCapacity = value;
         else Console.Error.WriteLine($"Warning: Car capacity {value} is too small");
      }
   }

   public static string ClownCarDescription()
   {
      return $"A clown car with {Random.Shared.Next(CarCapacity / 2, CarCapacity)} clowns";
   }
}
```

Add this!

The static carCapacity field is marked with the `protected` access modifier. That means it can only be accessed from within IClown or any interface that *extends* IClown (such as IScaryClown).

Now you can update the top-level statements to access the static IClown members:

```
IClown.CarCapacity = 18;
Console.WriteLine(IClown.ClownCarDescription());
IClown fingersTheClown = new ScaryClown("a big red nose", 14);
fingersTheClown.Honk();
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareLittleChildren();
}
```

Try adding a *private field* to your interface. You can add one—but only if it's static! If you remove the `static` keyword, the compiler will tell you that interfaces can't contain instance fields.

These static interface members behave exactly like the static class members that you've used in previous chapters. Public members can be used from any class, private members can only be used from inside IClown, and protected members can be used from IClown or any interface that extends it.

Default implementations give bodies to interface methods

All of the methods that you've seen in interfaces so far—except for the static methods—have been abstract: they don't have bodies, so any class that implements the interface must provide an implementation for the method.

But you can also provide a **default implementation** for any of your interface methods. Here's an example:

```
interface IWorker {
    string Job { get; }
    void WorkTheNextShift();

    void Buzz() {
        Console.WriteLine("Buzz!");
    }
}
```

You can even add private methods to your interface if you want, but they can only be called from public default implementations.

You can call the default implementation—but you **must use an interface reference** to make the call:

```
IWorker worker = new NectarCollector();
worker.Buzz();
```

But this code will not compile—it will give you the error "NectarCollector' does not contain a definition for 'Buzz'":

```
NectarCollector pearl = new NectarCollector();
pearl.Buzz();
```

The reason is that when an interface method has a default implementation, that makes it a virtual method, just like the ones you used in classes. Any class that implements the interface has the option to implement the method. The virtual method is ***attached to the interface***. Like any other interface implementation, it's not inherited. That's a good thing—if a class inherited default implementations from every interface that it implemented, then if two of those interfaces had methods with the same name the class would run into the Deadly Diamond of Death.

Use @ to create verbatim string literals

The @ character has special meaning in C# programs. When you put it at the beginning of a string literal, it tells the C# compiler that the literal should be interpreted verbatim. That means slashes are not used for escape sequences—so @"\n" will contain a slash character and an n character, not a newline. It also tells the C# compiler to include any line breaks. So this: @"Line 1

Line 2" is the same as "Line1\nLine2" (including the line break).

You can use verbatim string literals to create multiline strings that include line breaks. They work great with string interpolation—just add a $ to the beginning.

Add a ScareAdults method with a default implementation

Our IScaryClown interface is state-of-the-art when it comes to simulating scary clowns. But there's a problem: it only has a method to scare little children. What if we want our clowns to terrify the living $#!* out of adults too?

We ***could*** add an abstract ScareAdults method to the IScaryClown interface. But what if we already had dozens of classes that implemented IScaryClown? And what if most of them would be perfectly fine with the same implementation of the ScareAdults method? That's where default implementations are really useful. A default implementation lets you add a method to an interface that's already in use **without having to update any of the classes that implement it**. Add a ScareAdults method with a default implementation to IScaryClown:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();

    void ScareAdults()
    {
        Console.WriteLine($@"I am an ancient evil that will haunt your nightmares.
Behold my necklace with {Random.Shared.Next(4, 10)} of my last victim's fingers.
Oh, also, before I forget...");
        ScareLittleChildren();
    }
}
```

We used a verbatim literal here. We could have used a normal string literal instead and added \n's for the line breaks. This way is a lot easier to read.

Add this!

Take a close look at how the ScareAdults method works. That method only has two statements, but there's a lot packed into them. Let's break down exactly what's going on:

- The `Console.WriteLine` statement uses a verbatim literal with string interpolation. The literal starts with `$@` to tell the C# compiler two things: the `$` tells it to use string interpolation, and the `@` tells it to use a verbatim literal. That means the string will include three line breaks.
- The literal uses string interpolation to call Random.Shared.Next(4, 10) to add an extra scary touch.
- We've seen throughout the book that when there's a static field, that means there's only one copy of that field. So there's just one copy of the protected static carCapacity field that all clowns share.
- The last line of the ScareAdults method calls ScareLittleChildren. That method is abstract in the IScaryClown interface, so it will call the version of ScareLittleChildren in the class that implements IScaryClown.
- That means ScareAdults will call the version of ScareLittleChildren that's defined in whatever class implements IScaryClown. ***Try implementing a new ScareLittleChildren method in ScaryClown.***

Call your new default implementation by modifying the block after the `if` statement in your top-level statements to call ScareAdults instead of ScareLittleChildren:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareAdults();
}
```

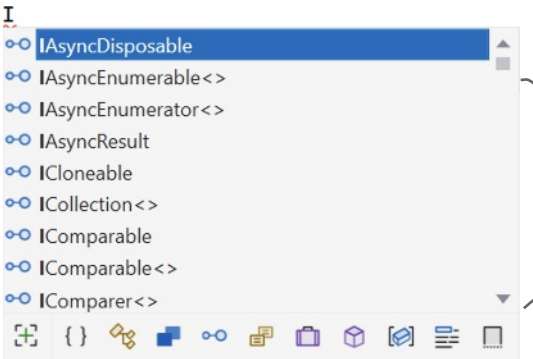
*Interfaces seem so **theoretical**. I can see how they work in the projects in this book, but do developers actually use them in real projects?*

C# developers use interfaces *all the time*, especially when we use libraries, frameworks, and APIs.

Developers always stand on the shoulders of giants. You're about halfway through this book, and in the first half you've written code that prints text to the console, draws windows with buttons, and renders 3D objects. You didn't need to write code to specifically output individual bytes to the console, or draw the lines and text to display buttons in a window, or do the math needed to display a sphere—you took advantage of code that other people wrote:

- You've used **frameworks** like .NET Core and .NET MAUI.
- You've used **APIs** like the Unity scripting API.
- The frameworks and APIs contain **class libraries** that you access with `using` directives at the top of your code.

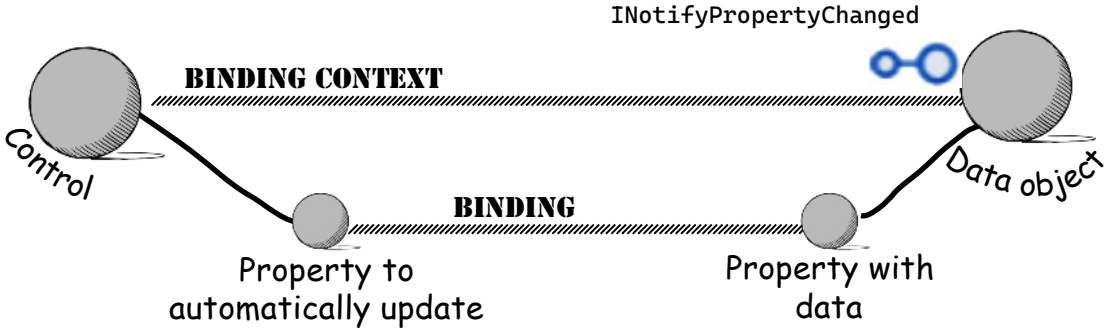
And when you're using libraries, frameworks, and APIs, you use interfaces a lot. See for yourself: open up a .NET Core or MAUI application, click inside of any method, and type `I` to pop up an IntelliSense window. Any potential match that has the symbol next to it is an interface. These are all interfaces that you can use to work with the framework.



These are just a few of the interfaces that are part of .NET Core.

Data binding updates MAUI controls automatically

Here's a great example of a real-world use case for an interface: **data binding**. Data binding is a really useful feature in MAUI that lets you set up your controls so their properties are automatically set based on a property in an object, and when that property changes your controls' properties are automatically kept up to date.

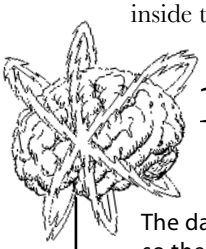


1. **Data binding starts with a *data object*.**
If you want your controls to automatically update themselves, they need to have some data to use. A **data object** is an object with data that you want to automatically send to your controls.

2. **The *binding context* tells your controls where to find the data.**
The goal of data binding is to feed data to the controls on your page automatically, so it makes sense that they would need some place to get that data. The binding context is the object that contains the data you'll use in your controls.

   You can set the binding context for a control by setting its **BindingContext property** to point to an object that it will get data from. If you set the binding context for a control that contains other controls—like a ScrollView or a Grid—then the nested controls get the same binding context.

3. **The XAML includes *bindings* that tell the controls what data to bind to.**
Once you have the data object created and the binding context set, you need to tell your controls to look for that data and use it to update themselves. You do this by setting up a **binding**, special text inside the XAML property value that looks like this: **`{Binding PropertyToBindTo}`**



Brain Power

The data object has a **specific job**: it needs to notify the binding context any time its data has changed, so the controls can update themselves. How do you think you'll tell your data object to do a specific job?

Add data binding to the default MAUI app

Let's use data binding to make the robot driving the race car in the default MAUI app tell you how it's feeling.

1 **Create a new .NET MAUI app and add a new class with data to bind to.**

Create a new .NET MAUI app, then **add a new class called Moods**. It has a property called CurrentMood that's updated any time its UpdateMood method is called:

Do this!

```
class Moods
{
    public string CurrentMood { get; private set; } = "I'm happy!";

    public void UpdateMood()
    {
        switch (Random.Shared.Next(1, 4))
        {
            case 1:
                CurrentMood = "I'm happy!";
                break;
            case 2:
                CurrentMood = "Oh so sad.";
                break;
            default:
                CurrentMood = "Feeling good.";
                break;
        }

        SemanticScreenReader.Announce(CurrentMood);
    }
}
```

When UpdateMood is called, it picks a random number from 1 to 3 and uses it to choose a mood.

Let's make it accessible! The screen reader will announce every mood change.

2 **Set the binding context for your page to a new instance of Moods.**

One easy way to set the binding context for all of the controls on your page is to set the BindingContext property of the page object in the MainPage constructor.

```
public partial class MainPage : ContentPage
{
    int count = 0;

    Moods moods = new Moods();

    public MainPage()
    {
        InitializeComponent();

        BindingContext = moods;
    }
```

Create a new instance of the Moods class.

Add a statement to the page's constructor to set its BindingContext property so it contains a reference to the Moods object.

3 **Add a new Label with a binding to the top of the page.**

Add this XAML for a Label control just below the opening `<VerticalStackLayout>` tag:

```
<Label Text="{Binding CurrentMood}" FontSize="Medium"
       HorizontalOptions="Center" Margin="20"/>
```

When you run your app, the Label binding to the CurrentMood looks up that property in the binding context, an instance of Moods.

④ **Run your app!**

Congratulations—your app now uses data binding! You created an instance of a data object, set the data context, and added a binding. Now the Label control uses that binding for its Text property. The binding tells it to read its data from the CurrentMood property in the data context.

⑤ **Update the mood when the button is clicked.**

The real power of data binding is being able to keep properties in the controls on your page up to date as the data object changes. **Modify the Click event handler method** to call the UpdateMood method on your Moods data object:

```
private void OnCounterClicked(object sender, EventArgs e)
{
      moods.UpdateMood();

      count++;

      if (count == 1)
            CounterBtn.Text = $"Clicked {count} time";
      else
            CounterBtn.Text = $"Clicked {count} times";

      SemanticScreenReader.Announce(CounterBtn.Text);
}
```

Calling UpdateMood will cause the CurrentMood to change to a new random mood every time the button is clicked.

⑥ **Run the app again.**

Hold on—something's not working. **The mood on the page isn't changing!** If you restart the app, it picks a new random mood. But clicking the button doesn't cause the page to get updated.

Try putting a breakpoint on the line that calls moods.UpdateMood and clicking the button. When the breakpoint breaks, hover over moods to see the value of the CurrentMood property:

Step over the method. The property value changes, but the page doesn't.

Why do you think the page doesn't update when the property changes?

Make Moods implement the INotifyPropertyChanged interface

A data object has a specific job: notifying the binding context any time its data has changed. Data objects can be any kind of object—they don't have to extend any specific class. We learned throughout this chapter that when you need an object to do a specific job, you make it implement an interface. In .NET MAUI data binding, data object is a job. So it makes sense that there would be an interface for it!

A data object that needs to update properties should implement the **INotifyPropertyChanged interface**. This interface does exactly what it sounds like—notifies the binding context any time a property has changed.

The INotifyPropertyChanged interface is in the System.ComponentModel namespace. When you added the *Moods.cs* file to your project, Visual Studio may have added `using` statements at the top. Add this `using` statement to the file if it isn't already there:

Do this!

```
using System.ComponentModel;
```

Then modify the Moods class to **implement the INotifyPropertyChanged interface**. The interface is not implemented, so Visual Studio will warn you about a compiler error:

```
3 references
class Moods : INotifyPropertyChanged
{
    4 references
    public str
    1 reference
    public Moo
    {
        UpdateMood();
    }
```

interface System.ComponentModel.INotifyPropertyChanged
Notifies clients that a property value has changed.

CS0535: 'Moods' does not implement interface member 'INotifyPropertyChanged.PropertyChanged'

Show potential fixes (Alt+Enter or Ctrl+.)

This compiler error happens when a class that implements an interface has not implemented all of its members.

Use the Quick Actions menu to implement the interface. Press Ctrl+. (Windows) or ⌘+. (macOS) to show potential fixes **and choose "Implement interface"** from the context menu.

```
3 references
class Moods : INotifyPropertyChanged
{
```

Implement interface
Implement all members explicitly

When you use the Quick Actions menu (Control . or ⌘ .) to implement an interface, Visual Studio *automatically generates* the interface members for you.

As soon as you choose that action from the menu, Visual Studio should add this line of code to your Moods class:

```
public event PropertyChangedEventHandler? PropertyChanged;
```

The INotifyPropertyChanged interface has a single member, an **event** called PropertyChanged.

We *haven't talked about events yet*. We'll give you all of the code that you need to use this event. You can learn more about events in our downloadable chapter on events and delegates: *https://github.com/head-first-csharp/fifth-edition*

Use the PropertyChanged event to make data binding work

This interface has a new keyword that we haven't shown yet: **`event`**. An event is a way for an object to ***let other objects know that something happened***. Buttons have a Clicked event, and you've written event handler methods for them: if you've hooked up an event handler method to a button, then when the user clicks the button, it **invokes** its Clicked event, which calls your event handler.

Now your Moods class has a PropertyChanged event, just like the Button class has a Clicked event.

Luckily, events are easy to use. **Add this method to your Moods class**:

Do this!

```
protected void OnPropertyChanged(string name)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Now all you need to do to tell the binding context that the CurrentMood property has changed is to call this method and pass the string **`"CurrentMood"`** as an argument to invoke the PropertyChanged event.

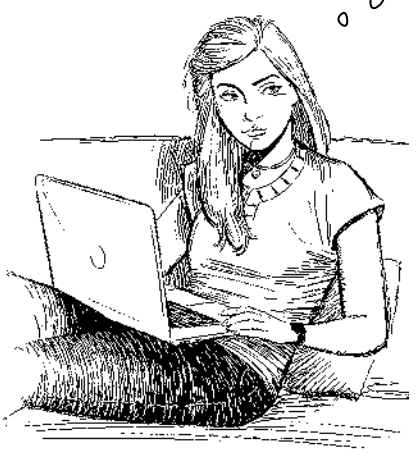
Modify the Moods class to **add this statement to the very end of the UpdateMood method**:

```
OnPropertyChanged("CurrentMood");
```

Now run your app again. Click the button several times. It now works—the label text updates.

You added an *event* to your class, and added a method that uses the *?. operator* to invoke the event to tell the binding context that a property was updated. That's all you need to know about events to finish the rest of the projects in this chapter!

*I see how data binding works, but I was fine updating the Text property on my labels. Do I **really need another way** to update my controls?*



Data binding can make your code-behind a lot simpler and easier to understand.

In Chapter 6 we talked about separation of concerns, and how you can make your code easier to read and work with if you separate behavior into different classes.

Data binding is a great way to use this idea to keep your code-behind simple, and to make it easier for you to write the code for your .NET MAUI apps—which can make a huge difference when you're building an app that has many controls and displays a lot of data.

Sharpen your pencil

In the next exercise, you'll convert the Beehive Management System to use data binding. Here's new code-behind for it. Compare it with the code in Chapter 6 and write down what we changed.

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        Dispatcher.StartTimer(TimeSpan.FromSeconds(1.5), TimerTick);

        BindingContext = queen;
    }

    private bool TimerTick()
    {
        if (!this.IsLoaded || !WorkShiftButton.IsVisible)
            return false;

        WorkShiftButton_Clicked(this, new EventArgs());
        return true;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        BindingContext = queen;
    }
}
```

What changed in the constructor?

.....

.....

.....

.....

.....

What changed in the rest of the code?

.....

.....

.....

Exercise

Modify the Beehive Management System to use data binding, using the code in the example that we just gave you.

Modify the Queen class to implement INotifyPropertyChanged

- Add the `using System.ComponentModel` directive to *Queen.cs* if it's not already there.
- Modify the Queen class declaration to add **`, INotifyPropertyChanged`** (including the comma) and use the Quick Action menu to implement the interface so it adds the PropertyChanged event.
- Add the OnPropertyChanged method to your Queen class (this is the same method that you added to *Moods.cs*).
- Add two new properties to Queen:

```
public bool HiveIsRunning { get; private set; } = true;
public bool OutOfHoney { get { return !HiveIsRunning; } }
```

- Add code to the end of the UpdateStatusReport method to notify the binding context that properties have changed:

```
OnPropertyChanged("StatusReport");
OnPropertyChanged("CanAssignWorkers");
OnPropertyChanged("HiveIsRunning");
OnPropertyChanged("OutOfHoney");
```

- Replace the last line of the WorkTheNext method with this code to update the HiveIsRunning property:

```
HiveIsRunning = base.WorkTheNextShift();
return HiveIsRunning;
```

Modify the XAML to bind property values to properties in the Queen class

Modify the XAML to bind AssignJobButton's IsEnabled property, WorkShiftButton's IsVisible property, OutOfHoneyButton's IsVisible property, and StatusReport's Text property to StatusReport.

```
<Button x:Name="AssignJobButton" Text="Assign this job to a bee"
        FontSize="Medium" Clicked="AssignJobButton_Clicked"
        SemanticProperties.Hint="Assigns the job to a worker bee"
        IsEnabled="{Binding CanAssignWorkers}" />

<Button x:Name="WorkShiftButton" Grid.Row="2" Text="Work the next shift"
        FontSize="Large" Clicked="WorkShiftButton_Clicked"
        SemanticProperties.Hint="Makes the bees work the next shift"
        IsVisible="{Binding HiveIsRunning}" />

<Button x:Name="OutOfHoneyButton" Grid.Row="2" Text="The hive is out of honey"
        BackgroundColor="Red" FontSize="Large" IsVisible="{Binding OutOfHoney}"
        Clicked="OutOfHoneyButton_Clicked"
        SemanticProperties.Hint="Restarts the Beehive Management System" />

<Label x:Name="StatusReport" Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"
       VerticalOptions="FillAndExpand" FontSize="Small"
       BackgroundColor="Black" TextColor="Yellow" Padding="10"
       Text="{Binding StatusReport}" />
```

The "Out of honey" button's IsVisible property is already set in the XAML, so make sure you change it.

Sharpen your pencil Solution

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        Dispatcher.StartTimer(TimeSpan.FromSeconds(1.5), TimerTick);

        BindingContext = queen;
    }

    private bool TimerTick()
    {
         if (!this.IsLoaded || !WorkShiftButton.IsVisible)
            return false;
        WorkShiftButton_Clicked(this, new EventArgs());
        return true;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        BindingContext = queen;
    }
}
```

The original Beehive Management System code reset the system by creating a new Queen object. We just need to update the binding context to get all of the page controls to reflect that change.

What changed in the constructor?

The constructor no longer calls the UpdateStatusAndEnableAssignButton method. Instead, it sets the binding context so it points to a reference to the Queen object in the queen field.

The updated code-behind in *MainPage.xaml.cs* is a lot shorter—and easier to understand!—because it doesn't have to manually set properties on controls.

What changed in the rest of the code?

UpdateStatusAndEnableAssignButton has been deleted. The code no longer sets the StatusReport.Text, AssignJobButton.IsEnabled, WorkShiftButton.IsVisible, or OutOfHoneyButton.IsVisible buttons. The OutOfHoneyButton_Clicked event handler method updates the binding context.

Exercise Solution

We gave you all of the changes to the XAML. Here are the changes to the Queen class. It's been modified to implement the INotifyPropertyChanged interface, and the OnPropertyChanged button was added.

```
class Queen : Bee, INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
```

This works exactly like the code that you added to the Moods class when you added the label to the default MAUI app.

Two new properties were added, which are used to bind to properties:

```
    public bool HiveIsRunning { get; private set; } = true;
    public bool OutOfHoney { get { return !HiveIsRunning; } }
```

The HiveIsRunning property defaults to true. If the hive runs out of honey, she flips it to false—which causes OutOfHoney to return true.

The UpdateStatusReport method was modified to call OnPropertyChanged every time the queen updated the report:

```
    private void UpdateStatusReport(bool allWorkersDidTheirJobs)
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"\nEgg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\n" +
        $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
        $"\n{WorkerStatus("Egg Care")}\nTOTAL WORKERS: {workers.Length}";

        if (!allWorkersDidTheirJobs)
            StatusReport += "\nWARNING: NOT ALL WORKERS DID THEIR JOBS";

        OnPropertyChanged("StatusReport");
        OnPropertyChanged("CanAssignWorkers");
        OnPropertyChanged("HiveIsRunning");
        OnPropertyChanged("OutOfHoney");
    }
```

These are the four properties that you added bindings to when you modified the XAML. Now the app will update them every time the status report is regenerated at the end of the shift.

The WorkTheNextShift method was modified to set the HiveIsRunning property before returning:

```
    public override bool WorkTheNextShift()
    {
        eggs += Constants.EGGS_PER_SHIFT;
        bool allWorkersDidTheirJobs = true;
        foreach (Bee worker in workers)
        {
            if (!worker.WorkTheNextShift())
                allWorkersDidTheirJobs = false;
        }
        HoneyVault.ConsumeHoney(unassignedWorkers * Constants.HONEY_PER_UNASSIGNED_WORKER);
        UpdateStatusReport(allWorkersDidTheirJobs);

        HiveIsRunning = base.WorkTheNextShift();
        return HiveIsRunning;
    }
```

All of the Bee subclasses call the base class's WorkTheNextShift method, which returns false if the hive is out of honey. This change uses that value to set the HiveIsRunning property before returning it.

there are no Dumb Questions

Q: I think I understand everything that we just did. Can you go over it again, just in case I missed something?

A: Absolutely. The Beehive Management System app you built in Chapter 6 updated its Label named StatusReport by setting its Text property in code like this:

```
StatusReport.Text = queen.StatusReport;
```

You modified that app to use data binding to automatically update the Label any time the Queen object updates its StatusReport property. You did this by making three changes. First, you modified the Queen class to implement the INotifyPropertyChanged interface so it could notify the UI of any changes to the property. Then you modified the XAML to create an instance of Queen and bind the Label.Text property to the Queen object's StatusReport property. Finally, you modified the code-behind to use the instance created by the XAML and remove the lines that set StatusReport.Text.

Q: And what exactly is the interface for?

A: The INotifyPropertyChanged interface gives you a way to tell a MAUI page that a property changed, so it can update any controls that bind to it. When you implement it, you're building a class that can do a specific job: notifying MAUI apps of property changes. The interface has a single member, an event called PropertyChanged. When you use your class for data binding, MAUI checks to see if it implements INotifyPropertyChanged, and if it does, it attaches an event handler to your class's PropertyChanged event, just like you attached event handlers to your Buttons' click events.

Q: I still don't quite get events. Can you explain them one more time?

A: Sure! You've been using **event handler methods** since the very first chapter in this book—for example, every time you used a button in a MAUI app, you added a Clicked event handler that looked like this:

```
private void Button_Clicked(object sender,
EventArgs e)
```

So what's going on behind the scenes? Well, a Button is actually just another class. Try going to the *MainPage.xaml.cs* file in the Beehive Management System and typing in **`WorkShiftButton`**. Hover over it:

```
WorkShiftButton
```

(field) Button MainPage.WorkShiftButton

Visual Studio is showing you that this is really just a field, and its type is Button. Somewhere inside the code for that Button is an event that's declared with the `event` keyword, and somewhere else there's code to invoke the event, just like your OnPropertyChanged method invoked the PropertyChanged event.

The reason the method is called an ***event*** handler method is because it handles an event—which means that it executes every time that event is invoked. Your MAUI's binding context does the same thing. It hooked up an event handler to the Queen's PropertyChanged event, and every time the Queen calls OnPropertyChanged to invoke the event, it updates the specific binding that you told it to update.

Watch it!

Data binding works with properties, not fields.

*You can only use data binding with **public properties**. If you try to bind a control attribute to a public field instead, you won't see any changes—but you won't get an exception either.*

Polymorphism means that one object can take many different forms

Any time you use a RoboBee in place of an IWorker, or a Wolf in place of an Animal, or even an aged Vermont cheddar in a recipe that just calls for cheese, you're using **polymorphism**. That's what you're doing any time you upcast or downcast. It's taking an object and using it in a method or a statement that expects something else.

You're using polymorphism when you take an instance of one class and use it in a statement or a method that expects a different type, like a parent class or an interface that the class implements.

Keep your eyes open for polymorphism!

You've been using polymorphism throughout—we just didn't use that word to describe it. While you're writing code over the next few chapters, be on the lookout for the many different ways you use it.

Here's a list of four typical ways that you'll use polymorphism. We're providing an example of each of them, though you won't see these particular lines in the exercises. As soon as you write similar code in an exercise in later chapters in the book, come back to this page and **check it off the following list**:

- ☐ **Taking any reference variable that uses one class and setting it equal to an instance of a different class.**

```
NectarStinger bertha = new NectarCollector();
IWorker genericBee = bertha;
```

- ☐ **Upcasting by using a subclass in a statement or method that expects its base class.**

```
spot = new Dog();
zooKeeper.FeedAnAnimal(spot);
```

If FeedAnAnimal expects an Animal object, and Dog inherits from Animal, then you can pass Dog to FeedAnAnimal.

- ☐ **Creating a reference variable whose type is an interface and pointing it to an object that implements that interface.**

```
IWorker defender = new HoneyManufacturer();
```

This is upcasting too!

- ☐ **Downcasting using the "is" keyword.**

```
if (worker is EggCare eggCareWorker)
    WorkNightShift(eggCareWorker);
```

The WorkNightShift method requires an EggCare reference, so we're using the "is" keyword to cast a generic IWorker reference back to an EggCare.

We've talked about **so many ways** to think about interfaces. Which one is the right way to think about them?

They're all right. Thinking about interfaces in lots of different ways helps you get to that *"aha!" moment*.

We've described interfaces in a bunch of different ways. You can think about an interface like:

- A **type** that you can use to reference any object that implements the interface
- A way to show that different classes do the same **job**
- A **contract** that guarantees that an object has certain members, no matter what its type is
- A **checklist** that the compiler uses to make sure that a class has all of the methods and properties in the interface

These are all good ways to think about interfaces

I bet we can use AI to help us to really understand how interfaces work—especially if we ask it to show us **lots of code examples** that illustrate all of these ideas.

Exactly! The fastest way to get to that "aha!" moment is to write lots of code that uses interfaces—and AI can help.

Interfaces are really useful, which is why there are so many different ways that C# developers use them. Ironically, that's one reason that the interface is one of the more difficult concepts for folks learning C# to wrap their heads around…so even if you feel like you've got a pretty good handle on the mechanics—the "***hows***"—of using interfaces, don't feel bad if you're still a little unclear on the "***whys***." Luckily, AI chatbots can be a great tool to help fill in your knowledge gaps and get you to that "aha!" moment when you really get interfaces.

Sens-AI

Use prompt engineering to learn more about interfaces

You've learned a lot about interfaces in this chapter! But even with the knowledge and practice you've had so far, you still have questions…right? Let's use AI to help you reach that "aha!" moment where you "get" interfaces.

OpenAI—the company that makes ChatGPT—created this really useful **prompt engineering guide** to help you use AI chatbots more effectively: *https://platform.openai.com/docs/guides/prompt-engineering*

Let's use some of the valuable advice in that guide to help get C# interfaces into your brain.

Prompt engineering strategy: Write clear instructions

The prompt engineering guide has **strategies** to help you get better results from an AI. You've already seen one of those strategies in action: ***write clear instructions***. Back in Chapter 3, you used the Guy object exercise from this book as a prompt. If the AI generated code that worked, that meant the instructions were clear enough for it.

We created the first version of that exercise in 2007 when we were writing the first edition of this book, and we've been revising and improving it ever since. We work really hard on writing *clear instructions* for each exercise in this book—if they're not clear, some readers have trouble with the exercise. Before AI, we had to test our instructions with real people (and we still do). Now we can test our instructions to see if they're clear by pasting them into an AI chatbot: if the chatbot generates code that isn't right, it means we didn't make our instructions clear enough.

Tactic: Ask the model to adopt a persona

Each strategy in the prompt engineering guide comes with tactics that you can use. This is one of them.

Asking an AI chatbot to adopt a **persona** means requesting that it change its responses to you in specific ways—like asking it to answer every prompt with a joke, or to talk like a pirate ("Ahoy, matey! Arrrr!"). Let's take advantage of this tactic by first giving it a prompt telling it how you want it to answer future questions, and then asking it the questions that you want answered. Here's a persona prompt that can help you when you're doing research:

```
When I ask for help to learn about a specific feature of C#, give a
response that has at least one real-world example I would use it for, with
lots of analogies and C# code that demonstrates that example.
```

Try it out. Start a new session with your favorite AI chatbot. Give it that prompt. It might choose a C# topic and give you an example of a response—that's OK. Once it finishes responding, ask it to help you learn about interfaces:

```
I want to learn more about C# interfaces. Can you help me understand how
an interface is like a contract?
```

Now it should include analogies and code examples in its response. Try asking it follow-up questions about interfaces, like how an interface is a type, or about how a class implements an interface to show that it does a specific job.

Tactic: Provide examples

Here's another tactic from the guide to help you write clear instructions.

AI chatbots are good at following examples. If you want to see the code first, then an explanation, and then an analogy, try giving it this prompt:

```
When you answer questions, follow this example:
Code: (add code here)
Explanation: (add explanation here)
Analogy: (add analogy here)
```

This prompt gives the AI chatbot an example of how you want it to answer questions in the future.

OK, I think I've got a **pretty good handle** on objects now!

The idea that you could combine your data and your code into classes and objects was a revolutionary one when it was first introduced—but that's how you've been building all your C# programs so far, so you can think of it as just plain programming.

You're an object-oriented programmer.

There's a name for what you've been doing. It's called **object-oriented programming**, or OOP. Before languages like C# came along, people didn't use objects and methods when writing their code. They just used functions (which is what they call methods in a non-OO program) that were all in one place—as if each program were just one big static class that only had static methods. It made it a lot harder to create programs that modeled the problems they were solving. Luckily, you'll never have to write programs without OOP, because it's a core part of C#.

The four core principles of object-oriented programming

When programmers talk about OOP, they're referring to four important principles. They should seem very familiar to you by now because you've been working with every one of them. We just told you about polymorphism, and you'll recognize the first three principles from Chapters 5 and 6: **inheritance**, **abstraction**, and **encapsulation**.

"Encapsulation" means creating an object that keeps track of its state internally using private fields, and uses public properties and methods to let other classes work with only the part of the internal data that they need to see.

This just means having one class or interface that inherits from another.

Inheritance

Encapsulation

Abstraction

You're using abstraction when you create a class model that starts with more general—or abstract—classes, and then has more specific classes that inherit from it.

The word "polymorphism" literally means "many forms." Can you think of a time when an object has taken on many forms in your code?

Polymorphism

Separation of concerns and DRY (don't repeat yourself) are also important ideas for OOP.

8 enums and collections

Organizing your data

Data isn't always as neat and tidy as you'd like it to be.

In the real world, you don't receive your data in tidy little bits and pieces. No, your data's going to come at you in loads, piles, and bunches. You'll need some pretty powerful tools to organize all of it—and lucky for us, C# has just the tools we need. **Enums** are types that let you define valid values to categorize your data. **Collections** are special objects that store many values, letting you **store, sort, and manage** all the data that your apps need to pore through. That way, you can spend your time thinking about writing apps to work with your data, and let the collections worry about keeping track of it for you.

If a constructor just sets fields, use a primary constructor instead

Let's take another look at the FunnyClown class from Chapter 7. It uses a constructor to set a private field and a read-only property that exposes that field to other classes. You added a constructor that takes a string argument and uses it to set a backing field for a read-only property.

```
class FunnyClown : IClown
{
    public FunnyClown(string funnyThingIHave)
    {
        this.funnyThingIHave = funnyThingIHave;
    }

    private string funnyThingIHave;

    public string FunnyThingIHave { get { return funnyThingIHave; } }

    public void Honk()
    {
        Console.WriteLine($"Hi kids! I have {funnyThingIHave}.");
    }
}
```

All this constructor does is use its argument to set a private field. This is a really common pattern for constructors.

A backing field and a read-only property is good encapsulation.

This is a good example of encapsulation. It's also very common—you'll frequently want a constructor that sets private fields. In fact, this is such a common pattern that C# has a built-in way to simplify it.

A **primary constructor** is a constructor that creates variables that you can use anywhere in the class. To create a primary constructor, just modify the class declaration to add arguments in parentheses after the class name.

Here's the class declaration for the FunnyClown class with a primary constructor added to it:

```
class FunnyClown(string funnyThingIHave) : IClown
```

A primary constructor for FunnyClown.

Since the original constructor just used its parameter to set the backing field, you can ***delete it entirely***. You don't need a backing field anymore either—your FunnyThingIHave read-only property can use the funnyThingIHave variable from the primary constructor.

Here's the complete FunnyClown class that uses a primary constructor:

```
class FunnyClown(string funnyThingIHave) : IClown
{
    public string FunnyThingIHave { get { return funnyThingIHave; } }

    public void Honk()
    {
        Console.WriteLine($"Hi kids! I have {funnyThingIHave}.");
    }
}
```

The FunnyThingIHave property uses the variable created by the parameterless constructor, so the class doesn't need a backing field at all.

Go back to your project from Chapter 7 and replace the FunnyClown class with this new one that uses a primary constructor. Your app should still work. You can create an instance of FunnyClown like this:

```
var bingoTheClown = new FunnyClown("ridiculously large shoes");
```

You'll get *lots of practice* with primary constructors throughout this chapter.

A primary constructor can extend a base constructor

Flip back to your Beehive Management System project in Chapter 6. Your EggCare class was a subclass of Bee, and its constructor used the `base` keyword to call the Bee constructor:

```
class EggCare(Queen queen) : Bee
{
    private Queen queen;

    public EggCare(Queen queen) : base("Egg Care")
    {
        this.queen = queen;
    }
    // ... the rest of the EggCare class ...
}
```

The EggCare constructor just sets a private field. That's a perfect job for a primary constructor! But it needs to call the Bee base class constructor. What do we do about that?

You can simplify the EggCare class using a primary constructor. When a primary constructor needs to call a base class constructor, **use the parameter in the class declaration** to pass either its own arguments or values.

Here's the simplified EggCare class:

```
class EggCare(Queen queen) : Bee("Egg Care")
{
    // ... the rest of the EggCare class ...
}
```

Overloaded constructors use this to call the primary constructor

Remember the PaintballGun class from Chapter 5? It has a "Balls" property with a private backing field called "balls." That's a great opportunity to add a primary constructor to simplify the class, but the app still needs that constructor with three parameters that sets MagazineSize and calls Reload if the `loaded` parameter is true:

```
class PaintballGun
{
    private int balls

    public PaintballGun(int balls, int magazineSize, bool loaded)
    {
        this.balls = balls;
        MagazineSize = magazineSize;
        if (loaded) Reload();
    }

    // ... the rest of the PaintballGun class ...
}
```

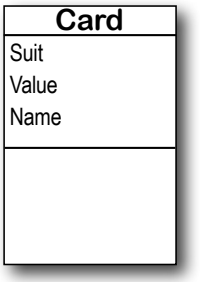
The solution is to add an **overloaded constructor that uses the `this` keyword** to call the primary constructor:

```
class PaintballGun(int balls)
{
    public PaintballGun(int balls, int magazineSize, bool loaded) : this(balls)
    {
        MagazineSize = magazineSize;
        if (loaded) Reload();
    }

    // ... the rest of the PaintballGun class ...
}
```

The original PaintballGun constructor is now an overloaded constructor that uses the `this` keyword to call the primary constructor first.

Strings don't always work for storing categories of data

We're going to be working with playing cards over the next few chapters, so let's build a Card class that we'll use. First, **create a new Card class with a primary constructor** that lets you pass it a suit and value, which it stores as strings:



This Card class uses string properties for suits and values.

```
class Card (string value, string suit)
{
    public string Value { get { return value; } }
    public string Suit { get { return suit; } }
    public string Name { get { return $"{Value} of {Suit}"; } }
}
```

That looks pretty good. We can create a Card object and use it like this:

```
Card aceOfSpades = new Card("Ace", "Spades");
Console.WriteLine(aceOfSpades.Name); // prints Ace of Spades
```

But there's a problem. Using strings to hold suits and values can have some unexpected results:

```
Card duchessOfRoses = new Card("Duchess", "Roses");
Card fourteenOfBats = new Card("Fourteen", "Bats");
Card dukeOfOxen = new Card("Duke", "Oxen");
```

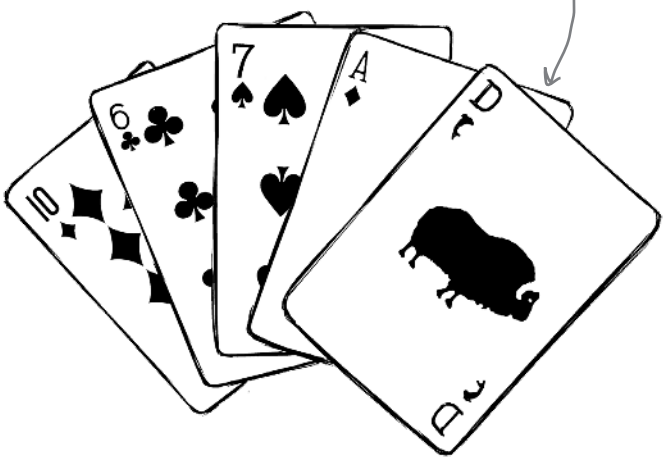
This code compiles, but these suits and values don't make any sense at all. The Card class really shouldn't allow these types as valid data.

We ***could*** add code to the constructor to check each string and make sure it's a valid suit or value, and handle bad input by throwing an exception. That's a valid approach—assuming you deal with the exceptions correctly, of course.

But ***wouldn't it be great*** if the C# compiler could automatically detect those invalid values for us? What if the compiler could ensure that all of the cards are valid before you even run the code? Well, guess what: it ***will*** do that! All you need to do is **enumerate** the values that are OK to use.

The rarely played Duke of Oxen card.

e-nu-me-rate, verb.
to specify one after another.
Ralph kept losing track of his pigeons, so he decided to ***enumerate*** *them by writing each of their names on a piece of paper.*



Enums let you work with a set of valid values

An **enum** or **enumeration type** is a data type that only allows certain values for that piece of data. So we could define an enum called `Suits`, and define the allowed suits:

```
enum Suits
{
    Diamonds,
    Clubs,
    Hearts,
    Spades,
}
```

Every enum starts with the `enum` keyword followed by its name. This enum is called `Suits`.

The rest of the enum contains a set of *members* inside a pair of curly braces, *separated by commas*. There's one member for each unique value—in this case, a member for each suit.

The last enum member isn't required to have a comma after it, but using one makes it easier to rearrange them using cut and paste.

An enum defines a new type

When you use the `enum` keyword you're **defining a new type**. Here are a few useful things to know about enums:

- **You can use an enum as the type in a variable definition, just like you'd use string, int, or any other type:**

```
Suits mySuit = Suits.Diamonds;
```

- **Since an enum is a type, you can use it to create an array:**

```
Suits[] myVals = new Suits[3] { Suits.Spades, Suits.Clubs, mySuit };
```

- **Use == to compare enum values. Here's a method that takes a Suits enum as a parameter, and uses == to check if it's equal to Suits.Hearts:**

```
void IsItAHeart(Suits suit)
{
    if (suit == Suits.Hearts) {
        Console.WriteLine("You pulled a heart!");
    } else {
        Console.WriteLine($"You didn't pull a heart: {suit}");
    }
}
```

An enum's *ToString method* returns the equivalent string, so Suits.Spades.ToString returns "Spades".

- **You can't just make up a new value for the enum. If you do, the code won't compile—which means you can avoid some annoying bugs:**

```
IsItAHeart(Suits.Oxen);
```

The compiler gives you an error if you use a value that's not part of the enum:

```
CS0117  'Suits' does not contain a definition for 'Oxen'
```

An enum lets you define a new type that only allows a specific set of values. Any value that's not part of the enum will break the code, which can prevent bugs later.

Enums let you represent numbers with names

Sometimes it's easier to work with numbers if you have names for them. You can assign numbers to the values in an enum and use the names to refer to them. That way, you don't have a bunch of unexplained numbers floating around in your code. Here's an enum to keep track of the scores for tricks at a dog competition:

```
enum TrickScore {
    Sit = 7,
    Beg = 25,
    RollOver = 50,
    Fetch = 10,
    ComeHere = 5,
    Speak = 30,
}
```

Members don't have to be in any particular order, and you can give multiple names to the same number.

Supply a name, then "=", then the number that name stands in for.

You can *cast an int to an enum*, and you can cast an (int-based) enum back to an int. Some enums use a different type, like byte or long (like the one below). You can cast those to their type instead of int.

Here's an excerpt from a method that uses the `TrickScore` enum by casting it to and from an int value:

```
int score = (int)TrickScore.Fetch * 3;
// The next line prints: The score is 30
Console.WriteLine($"The score is {score}");
```

The (int) cast tells the compiler to turn this into the number it represents. So since TrickScore.Fetch has a value of 10, (int) TrickScore.Fetch turns it into the int value 10.

You can cast the enum as a number and do calculations with it. You can even convert it to a string—an enum's ToString method returns a string with the member name:

```
TrickScore whichTrick = (TrickScore)7;
// The next line prints: Sit
Console.WriteLine(whichTrick.ToString());
```

You can cast an int back to a TrickScore, and TrickScore.Sit has the value 7.

Console.WriteLine calls the enum's ToString method, which returns a string with the member name.

If you don't assign any number to a name, the items in the list will be given values by default. The first item will be assigned a 0 value, the second a 1, etc. But what happens if you want to use really big numbers for one of the enumerators? The default type for the numbers in an enum is int, so you'll need to specify the type you need using the colon (:) operator, like this:

```
enum LongTrickScore : long {
        Sit = 7,
        Beg = 2500000000025
}
```

This tells the compiler to treat values in the TrickScore enum as longs, not ints.

This number is too big to fit into an int.

If you tried to use this enum without specifying long as the type, you'd get an error:

CS0266 Cannot implicitly convert type 'long' to 'int'.

Exercise

Card
Value Suit Name

Use what you've learned about enums and primary constructors to build a class that holds a playing card. Start by creating a **new Console App project** and adding a class called Card.

Add two public properties to Card: Suit (which will be Spades, Clubs, Diamonds, or Hearts) and Value (Ace, Two, Three…Ten, Jack, Queen, King). You'll also need one more property: a public read-only property called Name that returns a string like "Ace of Spades" or "Five of Diamonds."

Add two enums to define the suits and values in their own *.cs files

Add each enum. In Windows, use the familiar Add >>Class feature, then **replace `class` with `enum`** in each file. In macOS, use Add>>New File and choose Empty Enumeration. **Use the Suits enum that we just showed you**, then create an enum for values. Make the values equal to their face values: (int)Values.Ace should equal 1, Two should be 2, Three should be 3, etc. Jack should equal 11, Queen should be 12, and King should be 13.

To add an enum in VSCode, add a file and choose the "Enum" template.

Add a primary constructor and Name property that returns a string with the name of the card

Add a primary constructor that takes two parameters, a Suit and a Value:

```
Card myCard = new Card(Values.Ace, Suits.Spades);
```

Add read-only properties called Value and Suit that return the value and suit passed into the primary constructor. Add a read-only property called Name with a getter that returns a string that describes the card. For example, this code:

```
Console.WriteLine(myCard.Name);
```

should print the following:

```
Ace of Spades
```

Make the top-level statements print the name of a random card

You can get your app to create a card with a random suit and value by casting a random number between 0 and 3 as a Suits enum and another random number between 1 and 13 as a Values enum. To do this, you can take advantage of a feature Random that gives it three different ways to call its Next method:

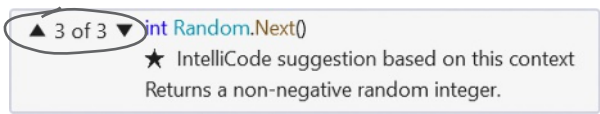
```
int numberBetween0and3 = Random.Shared.Next(4);
int numberBetween1and13 = Random.Shared.Next(1, 14);
int anyRandomInteger = Random.Shared.Next();
```

When you've got more than one way to call a method, it's called *overloading*. Hover over each of these overloaded `Next` methods in Visual Studio to see an explanation of how they work.

there are no Dumb Questions

Q: I remember calling Random.Shared.Next with two arguments earlier in the book. I noticed when I called the method, an IntelliSense window popped up that said "3 of 3" in the corner. Does that have to do with overloading?

A: Yes! When a class has an **overloaded** method—or a method that you can call more than one way—the IDE shows all of the options you have. In this case, the Random class has three possible Next methods. As soon as you type **`Random.Shared.Next(`** into the code window, the IDE pops up its IntelliSense box that shows the parameters for the different overloaded methods. The up and down arrows next to the "3 of 3" let you scroll between them. That's really useful when you're dealing with a method that has dozens of overloaded definitions. So when you're calling Random.Next, make sure you choose the right overloaded method. But don't worry too much now—we'll talk a lot about overloading later on in the chapter.



Exercise Solution

A deck of cards is a great example of a situation where limiting values is important. Nobody wants to turn over their cards and be faced with a 28 of Hearts or Ace of Hammers. Here's our Card class—you'll **reuse it a few times** over the next few chapters, so ***make sure your code matches the solution to this exercise***.

The Suits enum is in a file called *Suits.cs*. You already have the code for it—it's identical to the Suits enum that we showed you earlier in the chapter. The Values enum is in a file called *Values.cs*. Here's its code:

```
enum Values {
    Ace = 1,
    Two = 2,
    Three = 3,
    Four = 4,
    Five = 5,
    Six = 6,
    Seven = 7,
    Eight = 8,
    Nine = 9,
    Ten = 10,
    Jack = 11,
    Queen = 12,
    King = 13,
}
```

Here's where we set the value of Values.Ace to 1...

...and the value of Values.King is 13.

We chose the names "Suits" and "Values" for the enums, while the properties in the Card class that use those enums for types are called Suit and Value. ***What do you think about these names?*** **Look at the names of other enums that you'll see throughout the book. Would "Suit" and "Value" make better names for these enums?**

There's no right or wrong answer—in fact, Microsoft's C# language reference page for enums has both singular (`Season`) and plural (`Days`) names. Take a minute and look at this page:

https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/enum

The Card class has a constructor that sets its Suit and Value properties, and a Name property that generates a string description of the card:

```
class Card (Values value, Suits suit) {

    public Values Value { get { return value; } }

    public Suits Suit { get { return suit; } }

    public string Name {
        get { return $"{Value} of {Suit}"; }
    }

}
```

The primary constructor creates private variables called value and suit, so we need these read-only properties to expose them—which makes sense for a playing card, because a card can never change its suit or value.

The Name property's get accessor takes advantage of the way an enum's ToString method returns its name converted to a string.

The top-level statements use a static Random reference to cast **`Suits`** and **`Values`** to instantiate a random Card. The first statement uses the overloaded Random.Shared.Next method that we showed you to generate a random number from 1 to 13, which gets cast to a Values value, and then does the same for a number from 1 to 4 that gets cast to a Suits value:

```
Card card = new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4));
Console.WriteLine(card.Name);
```

We could use an array to create a deck of cards...

What if you wanted to create a class to represent a deck of cards? It would need a way to keep track of every card in the deck, and it'd need to know what order they were in. A Cards array would do the trick—the top card in the deck would be at value 0, the next card at value 1, etc. Here's a starting point—a Deck that starts out with a full deck of 52 cards:

```
class Deck
{
    private readonly Card[] cards = new Card[52];

    public Deck() {
        int index = 0;
        for (int suit = 0; suit <= 3; suit++)
        {
            for (int value = 1; value <= 13; value++)
            {
                cards[index++] = new Card((Values)value, (Suits)suit);
            }
        }
    }

    public void PrintCards()
    {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name);
    }
}
```

We used two for loops to iterate through all of the possible suit and value combinations.

...but what if you wanted to do more?

Think of everything you might need to do with a deck of cards, though. If you're playing a card game, you routinely need to change the order of the cards, and add and remove cards from the deck. You just can't do that with an array very easily. For example, take another look at the AddWorker method from the Beehive Management System exercise in Chapter 6:

```
private void AddWorker(Bee worker) {
    if (unassignedWorkers >= 1) {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
```

You used this code to add an element to an array in Chapter 6. What would you need to do if you want to add the Bee reference to the middle of the array instead of the end?

You had to use Array.Resize to make the array longer, then add the worker to the end. That's a lot of work! There must be an easier way to handle that. Can we do better?

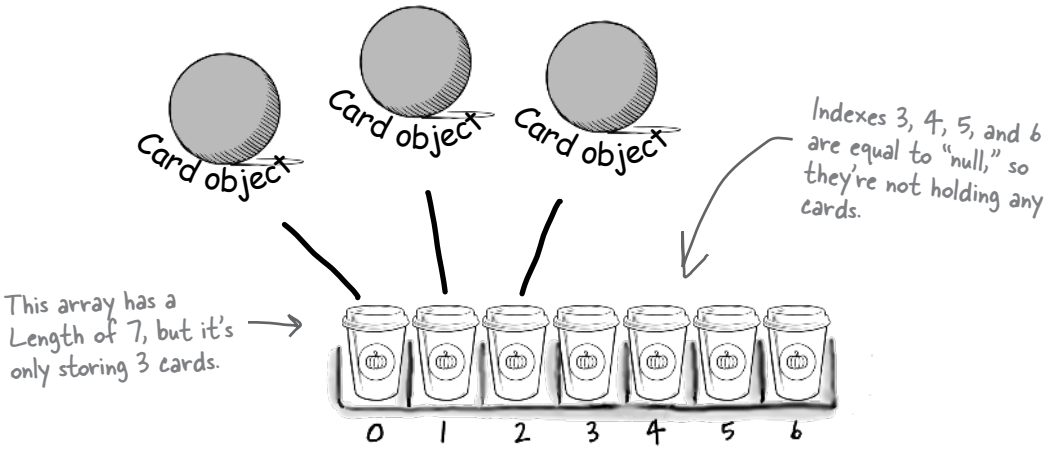
Brain Power

How would you add a Shuffle method to the Deck class that rearranges the cards in random order? What about a method to deal the first card off the top of the deck that returns it and then removes it from the deck? How would you add a card to the deck?

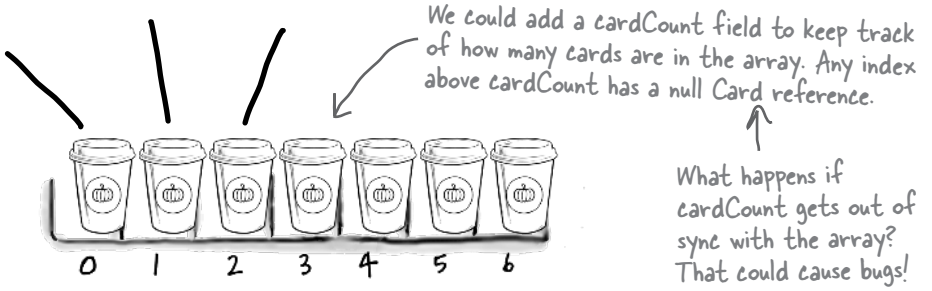
Arrays can be annoying to work with

An array is fine for storing a fixed list of values or references. Once you need to move array elements around, or add more elements than the array can hold, things start to get a little sticky. Here are a few ways that working with arrays can be troublesome.

Every array has a length. That length doesn't change unless you resize the array, so you need to know the length to work with it. Let's say you want to use an array to store Card references. If the number of references you want to store is less than the length of the array, you could use null references to keep some array elements empty.



You'd need to keep track of how many cards are being held in the array. You could add an int field—maybe you'd call it cardCount—that would hold the index of the last card in the array. So your three-card array would have a Length of 7, but you'd set cardCount equal to 3.



Now things get complicated. It's easy enough to add a Peek method that just returns a reference to the top card, so you can peek at the top of the deck. What if you want to add a card? If cardCount is less than the array's Length, you can just put your card in the array at that index and add 1 to cardCount. But if the array is full, you'll need to create a new, bigger array and copy the existing cards to it. Removing a card is easy enough—but after you subtract 1 from cardCount, you'll need to make sure to set the removed card's array index back to `null`. What if you need to **remove a card from the middle of the list**? If you remove card 4, you'll need to move card 5 back to replace it, and then move 6 back, then 7…***wow, what a mess!***

The AddWorker method from Chapter 6 used the Array.Resize method to do this.

Lists make it easy to store collections of...anything

C# and .NET have **collection** classes that handle all of those nasty issues that come up when you add and remove array elements. The most common sort of collection is a **List<T>**. Once you create a List<T> object, it's easy to add an item, remove an item from any location in the list, peek at an item, and even move an item from one place in the list to another. Here's how a list works.

We'll sometimes leave the <T> off when referring to List in the book. When you see List, think List<T>.

1. **First you create a new instance of List<T>.** Recall that every array has a type—you don't just have an array, you have an int array, a Card array, etc. Lists are the same. You need to specify the type of object or value that the list will hold by putting it in angle brackets (<>) when you use the `new` keyword to create it:

```
List<Card> cards = new List<Card>();
```

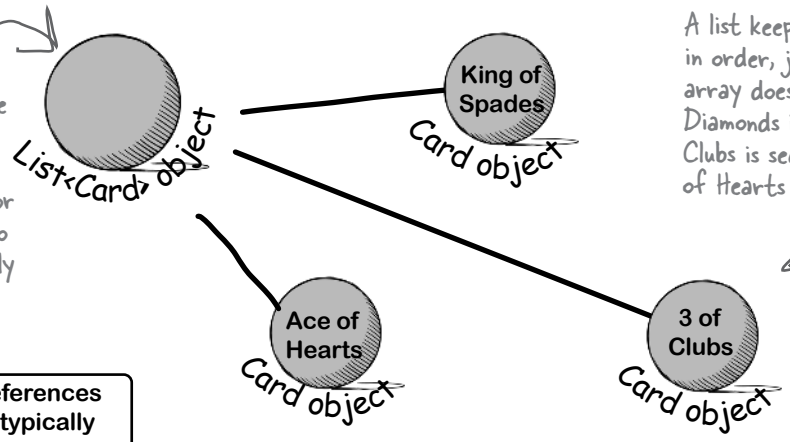
You specified <Card> when you created the list, so now this list only holds references to Card objects.

The <T> at the end of List<T> means it's *generic*. The T gets replaced with a type—so List<int> just means a List of ints. You'll get plenty of practice with generics over the next few chapters.

2. **Now you can add to your List<T>.** Once you've got a List<T> object, you can add as many items to it as you want as long as they're ***polymorphic*** with whatever type you specified when you created your new List<T>—which means they're assignable to the type (and that includes interfaces, abstract classes, and base classes).

```
cards.Add(new Card(Values.King, Suits.Spades));
cards.Add(new Card(Values.Three, Suits.Clubs));
cards.Add(new Card(Values.Ace, Suits.Hearts));
```

You can add as many Cards as you want to the List—just call its Add method. It'll make sure it's got enough "slots" for the items. If it starts to run out, it'll automatically resize itself.



A list keeps its elements in order, just like an array does. King of Diamonds is first, 3 of Clubs is second, and Ace of Hearts is third.

The values or object references contained in a list are typically referred to as its *elements*.

Lists are more flexible than arrays

The List class is built into .NET, and it lets you do a lot of things with objects that you can't do with a plain old array. Check out some of the things you can do with a List<T>.

`new List<Egg>();` creates a List of Egg objects. It starts out empty. You can add or remove objects, but since it's a List of Eggs, you can only add references to Egg objects, or to any object that can be cast to an Egg.

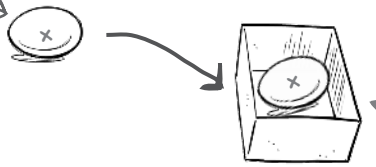
1. **Use the `new` keyword to instantiate a List (like you'd expect!)**

```
List<Egg> myCarton = new List<Egg>();
```

Here's a reference to an Egg object.

2. **Add something to a List.**

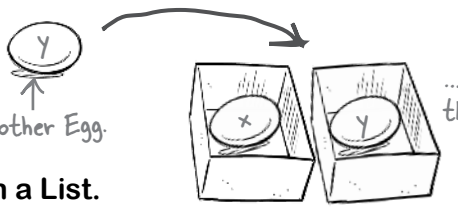
```
Egg x = new Egg();
myCarton.Add(x);
```



Now the List expands to hold the Egg object...

3. **Add something else to the List.**

```
Egg y = new Egg();
myCarton.Add(y);
```



Here's another Egg.

...and expands again to hold the second Egg object.

4. **Find out how many things are in a List.**

```
int theSize = myCarton.Count;
```

5. **Find out if your List has a *particular* thing in it.**

```
bool isIn = myCarton.Contains(x);
```

Now you can search for a specific Egg inside the List. This would definitely come back true because you just added that egg to your List.

x y

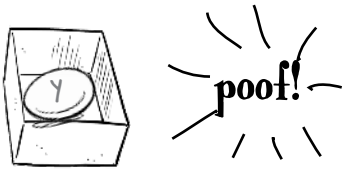
6. **Figure out where in the List that thing is.**

```
int index = myCarton.IndexOf(x);
```

The index for x would be 0 and the index for y would be 1.

7. **Take that thing out of the List.**

```
myCarton.Remove(x);
```



When we removed x, we left only y in the List, so it shrank! If we remove y, eventually it will get garbage-collected.

Sharpen your pencil

Fill in the rest of this table by looking at the List code on the left and entering what you think the code might be if it were using a regular array instead. We don't expect you to get all of them exactly right, so just make your best guess.

Here are a few lines from the middle of a program. Assume these statements are all executed in order, one after another, and that the variables were previously declared.

We filled in a few for you...

List	Regular array
`List<String> myList = new List <String>();`	String [] myList = new String[2];
`String a = "Yay!";`	String a = "Yay!";
`myList.Add(a);`	
`String b = "Bummer";`	String b = "Bummer";
`myList.Add(b);`	
`int theSize = myList.Count;`	
`Guy o = guys[1];`	
`bool foundIt = myList.Contains(b);`	

Hint: you'll need more than one line of code here.

Sharpen your pencil Solution

Your job was to fill in the rest of the table by looking at the List code on the left and entering what you think the code might be if it were using a regular array instead.

List	Regular array
`List<String> myList = new List <String>();`	`String[] myList = new String[2];`
`String a = "Yay!"`	`String a = "Yay!";`
`myList.Add(a);`	`myList[0] = a;`
`String b = "Bummer";`	`String b = "Bummer";`
`myList.Add(b);`	`myList[1] = b;`
`int theSize = myList.Count;`	`int theSize = myList.Length;`
`Guy o = guys[1];`	`Guy o = guys[1];`

`bool foundIt = myList.Contains(b);`	`bool foundIt = false;`
	`for (int i = 0; i < myList.Length; i++) {`
	`  if (b == myList[i]) {`
	`  foundIt = true;`
	`  }`
	`}`

Lists are objects that have methods, just like every other class you've used so far. You can see the list of methods available from within the IDE just by typing a **.** next to the List name, and you pass parameters to them just the same as you would for a class you created yourself.

The elements in a list are ordered, and the element's position in a list is called its **index**. Just like with an array, list indexes start at 0. You can access the element at a specific index of a list using its **indexer**:

```
Guy o = guys[1];
```

"Elements" is another name for the items in a list.

With arrays, you're a lot more limited. You need to set the size of the array when you create it, and you'll have to write any logic that needs to be performed on it on your own.

The Array class does have some static methods that make some of these things a little easier to do—for example, you already saw the Array.Resize method, which you used in your AddWorker method. But we're concentrating on List objects because they're a lot easier to use.

Let's build an app to store shoes

Use a List to write a console app that prompts the user to add or remove shoes. Here's an example of what it looks like to run the app, adding two shoes and then removing them:

We'll start with a Shoe class that stores the style and color for a shoe. Then we'll create a class called ShoeCloset that stores the shoes in a List<Shoe>, with AddShoe and RemoveShoe methods that prompt the user to add or remove shoes.

Do this!

1. **Add an enum for shoe style.** Some shoes are sneakers, others are sandals, so an enum makes sense:

```
enum Style
{
    Sneaker,
    Loafer,
    Sandal,
    Flipflop,
    Wingtip,
    Clog,
}
```

Remember from earlier that you can cast an enum to and from an int. So Sneaker is equal to 0, Loafer is 1, etc.

2. **Add the Shoe class.** It uses the `Style` enum for the shoe style and a string for shoe color, and works just like the Card class we created earlier in the chapter—with read-only getters to return the private variables created by the primary constructor—and it uses `string?` for color because it's reading console input:

```
class Shoe(Style style, string? color)
{
    public Style Style {
       get { return style; }
    }

    public string? Color {
       get { return color; }
    }

    public string Description
    {
        get { return $"A {Color} {Style}"; }
    }
}
```

```
The shoe closet is empty.

Press 'a' to add or 'r' to remove a shoe: a
Add a shoe
Press 0 to add a Sneaker
Press 1 to add a Loafer
Press 2 to add a Sandal
Press 3 to add a Flipflop
Press 4 to add a Wingtip
Press 5 to add a Clog
Enter a style: 1
Enter the color: black

The shoe closet contains:
Shoe #1: A black Loafer

Press 'a' to add or 'r' to remove a shoe: a
Add a shoe
Press 0 to add a Sneaker
Press 1 to add a Loafer
Press 2 to add a Sandal
Press 3 to add a Flipflop
Press 4 to add a Wingtip
Press 5 to add a Clog
Enter a style: 0
Enter the color: blue and white

The shoe closet contains:
Shoe #1: A black Loafer
Shoe #2: A blue and white Sneaker

Press 'a' to add or 'r' to remove a shoe: r
Enter the number of the shoe to remove: 2
Removing A blue and white Sneaker

The shoe closet contains:
Shoe #1: A black Loafer

Press 'a' to add or 'r' to remove a shoe: r
Enter the number of the shoe to remove: 1
Removing A black Loafer

The shoe closet is empty.

Press 'a' to add or 'r' to remove a shoe:
```

Press 'a' to add a shoe, then choose the type of shoe and type in the color.

Press 'r' to remove a shoe, then enter the number of the shoe to remove.

3 **The ShoeCloset class uses a List<Shoe> to manage its shoes.** The ShoeCloset class has three methods—the PrintShoes method prints a list of shoes to the console, the AddShoe method prompts the user to add a shoe to the closet, and the RemoveShoe method prompts the user to remove a shoe:

```
class ShoeCloset
{
    private readonly List<Shoe> shoes = new List<Shoe>();

    public void PrintShoes()
    {
        if (shoes.Count == 0)
        {
            Console.WriteLine("\nThe shoe closet is empty.");
        }
        else
        {
            Console.WriteLine("\nThe shoe closet contains:");
            int i = 1;
            foreach (Shoe shoe in shoes)
            {
                Console.WriteLine($"Shoe #{i++}: {shoe.Description}");
            }
        }
    }

    public void AddShoe()
    {
        Console.WriteLine("\nAdd a shoe");
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine($"Press {i} to add a {(Style)i}");
        }
        Console.Write("Enter a style: ");
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int style))
        {
            Console.Write("\nEnter the color: ");
            string? color = Console.ReadLine();
            Shoe shoe = new Shoe((Style)style, color);
            shoes.Add(shoe);
        }
    }

    public void RemoveShoe()
    {
        Console.Write("\nEnter the number of the shoe to remove: ");
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int shoeNumber) &&
            (shoeNumber >= 1) && (shoeNumber <= shoes.Count))
        {
            Console.WriteLine($"\nRemoving {shoes[shoeNumber - 1].Description}");
            shoes.RemoveAt(shoeNumber - 1);
        }
    }
}
```

Here's the List that contains the references to the Shoe objects.

ShoeCloset
private List<Shoe> shoes
PrintShoes
AddShoe
RemoveShoe

This "foreach" loop iterates through the "shoes" list and writes a line to the console for each shoe.

The "for" loop sets "i" to an integer from 0 to 5. The interpolated string uses {(Style)i} to cast it to a Style enum and then calls its ToString method to print the member name.

Here's where we create a new Shoe instance and add it to the list.

This is just like code you've seen before: it calls Console.ReadKey, then uses KeyChar to get the character that was pressed. int.TryParse needs a string, not a char, so we call ToString to convert the char to a string.

Here's where we remove a Shoe instance from the list.

Really take the time to *step through this code*, read our notes, and understand how it works.

4 **Add the top-level statements.** Notice how it doesn't do very much? That's because all of the interesting behavior is encapsulated in the ShoeCloset class:

```
ShoeCloset shoeCloset = new ShoeCloset();

while (true)
{
    shoeCloset.PrintShoes();
    Console.Write("\nPress 'a' to add or 'r' to remove a shoe: ");
    char key = Console.ReadKey().KeyChar;
    switch (key)
    {
        case 'a':
        case 'A':
            shoeCloset.AddShoe();
            break;
        case 'r':
        case 'R':
            shoeCloset.RemoveShoe();
            break;
        default:
            return;
    }
}
```

There's no "break" statement after the 'a' case, so it falls through to the 'A' case—so they're both handled by shoeCloset.AddShoe.

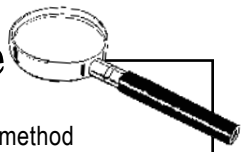
We used a `switch` statement to handle the user input. We wanted uppercase 'A' to work the same as lowercase 'a', so we included two `case` statements next to each other without a `break` between them:

```
case 'a':
case 'A':
```

When a `switch` encounters a new `case` statement without a `break` before it, it *falls through* to the next `case`. You can even have statements between the two `case` statements. Be really careful with this—it's easy to accidentally leave out a `break` statement.

5 **Run your app and reproduce the sample output.** Try debugging the app, and start to get familiar with how you work with lists. No need to memorize anything right now—you'll get plenty of practice with them!

List Class Members Up Close



The List collection class has an Add method that adds an item to the end of the list. The AddShoe method creates a Shoe instance, then calls the shoes.Add method, passing it a reference to that instance:

```
shoes.Add(shoe);
```

The List class also has a RemoveAt method that removes an item from a specific index in the list. Lists, like arrays, are **zero-indexed**, which means the first item has index 0, the second item has index 1, etc.:

```
shoes.RemoveAt(shoeNumber - 1);
```

And finally, the PrintShoes method uses the List.Count property to check if the list is empty:

```
if (shoes.Count == 0)
```

Generic collections can store any type

You've already seen that a list can store strings or Shoes. You could also make lists of integers or any other object you can create. That makes a list a **generic collection**. When you create a new list object, you tie it to a specific type: you can have a list of ints, or strings, or Shoe objects. That makes working with lists easy—once you've created your list, you always know the type of data that's inside it.

But what does "generic" really mean? Let's use Visual Studio to explore generic collections. Open *ShoeCloset.cs* and hover your mouse cursor over `List`:

```
private readonly List<Shoe> shoes = new List<Shoe>();
```

class System.Collections.Generic.List<T>
Represents a strongly typed list of objects that can be accessed by index.
Provides methods to search, sort, and manipulate lists.
T is Shoe

There are a few things to notice:

- The List class is in the namespace System.Collections.Generic—this namespace has several classes for generic collections (you don't need a `using` statement because the System.Collections namespace is automatically included with top-level statements).
- The description says that List provides "methods to search, sort, and manipulate lists." You used some of these methods in your ShoeCloset class.
- The top line says `List<T>` and the bottom says `T is Shoe`. This is how generics are defined—it's saying that List can handle any type, but for this specific list, that type is the Shoe class.

A generic collection can hold any type of object, and gives you a consistent set of methods to work with the objects in the collection no matter what type of object it's holding.

Generic lists are declared using <angle brackets>

When you declare a list—no matter what type it holds—you always declare it the same way, using <angle brackets> to specify the type of object being stored in the list.

You'll often see generic classes (not just List) written like this: List<T>. That's how you know the class can take any type.

This doesn't actually mean that you add the letter T. It's a notation that you'll see whenever a class or interface works with all types. The <T> part means you can put a type in there, like List<Shoe>, which limits its members to that type.

```
List<T> name = new List<T>();
```

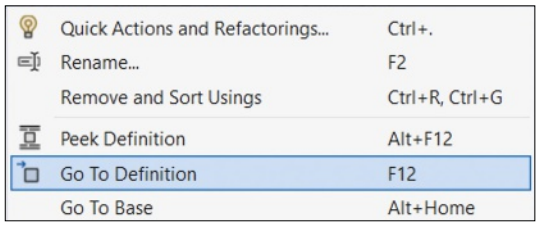
Lists can be either very flexible (allowing any type) or very restrictive. So they do what arrays do, and then quite a few things more.

ge-ne-ric, adjective.
characteristic of or relating to a class or group of things; not specific. "*Developer*" *is a* ***generic*** *term for anyone who writes code, no matter what kind of code they write.*

IDE Tip: Go to Definition

The List class is part of .NET, which has a whole bunch of really useful classes, interfaces, types, and more. Visual Studio has a really powerful tool that you can use to explore these classes, and any other code you've written. Open *Program.cs* and find this line: `ShoeCloset shoeCloset = new ShoeCloset();`

Right-click on ShoeCloset and choose **Go To Definition**:



You can click on almost anything in the IDE and press F12 to go to definition. You can also control-click (or ⌘-click on a Mac) to do the same thing. You can even go to the definition of classes, enums, or other members from .NET (like List<T>).

The IDE will jump straight to the definition of the ShoeCloset class. Now go back to *Program.cs* and go to the definition of PrintShoes in this line: `shoeCloset.PrintShoes();`. The IDE will jump straight to that method definition in the ShoeCloset class. You can use Go To Definition to quickly jump around your code.

Use Go To Definition to explore generic collections

Now comes the really interesting part. Open up *ShoeCloset.cs* and go to the definition of **`List`**. The IDE will open a separate tab with the definition of the List class. Don't worry if this new tab has a lot of complex stuff on it! You don't need to understand it all—just find this line of code, which shows you how List<T> implements a *bunch* of interfaces:

```
public class List<T> :  IList<T>, ... (there will be a bunch of interfaces that include IList<T>)
```

Now **go to the definition of IList<T>**. Notice how it implements ICollection<T>? That's the interface used by every generic collection. You probably guessed what you'll do next—go to the definition for ICollection<T>. Here's what you'll see in Visual Studio for Windows (the XML comments are collapsed and replaced with [...] buttons; they may be expanded on Mac):

Visual Studio Code may not let you go to definition from that window. You can get to the same place by adding an IList<string> field to a class and going to its definition.

```
namespace System.Collections.Generic
{
    [...]public interface ICollection<T> : IEnumerable<T>, IEnumerable
    {
        [...]int Count { get; }
        [...]bool IsReadOnly { get; }

        [...]void Add(T item);
        [...]void Clear();
        [...]bool Contains(T item);
        [...]void CopyTo(T[] array, int arrayIndex);
        [...]bool Remove(T item);
    }
}
```

A generic collection lets you find out how many items it has, add new items, clear it, check if it contains an item, and remove an item. It may do other things too—like a List, which lets you remove an item at a specific index—but any generic collection meets this minimum standard.

In Chapter 9, we talked about how interfaces are all about making classes do jobs. A generic collection is a specific job. Any class can do it, as long as it implements the ICollection<T> interface. The List<T> class does, and you'll see a few more collection classes later in the chapter that do. They all work a bit differently, but because they all do the job of being a generic collection, you can depend on them all doing the job of storing values or references.

Bullet Points

- **List** is a .NET class that lets you store, manage, and easily work with a set of values or object references. The values or references stored in a list are often referred to as its **elements**.
- A List resizes dynamically to whatever size is needed. As you add data to the List, it grows to accommodate it.
- To put something into a List, use **Add**. To remove something from a List, use **Remove**.
- You can remove objects from a List using their index number with **RemoveAt**.
- You declare the type of the List using a type argument, which is a type name in angle brackets. For example: List<Frog> means the List will be able to hold only objects of type Frog.
- You can use the **Contains method** to find out if a particular object is in a List. The **IndexOf** method returns the index of a specific element in the List.
- The **Count** property returns the number of elements in the list.
- Use an **indexer** (like `guys[3]`) to access the item in a collection at a specific index.
- You can use `foreach` **loops** to iterate through lists, just like you do with arrays.
- A List is a **generic collection**, which means it can store any type.
- All generic collections implement the generic **ICollection<T> interface**.
- The <T> in a generic class or interface definition is **replaced with a type** when you instantiate it.
- Use the **Go To Definition** (Windows) or **Go to Declaration** (macOS) feature in Visual Studio to explore your code and other classes that you use.

Watch it!

Don't modify a collection while you're using `foreach` to iterate through it!

If you do, it will throw an InvalidOperationException. You can see this for yourself. Create a new console app, then add code to create a new List<string>, add a value to it, use `foreach` *to iterate through it, and add another value to the collection inside the* `foreach` *loop. When you run your code, your* `foreach` *loop will throw an exception. And remember, you always specify a type when using generic classes—so List<string> refers to a list of strings.*

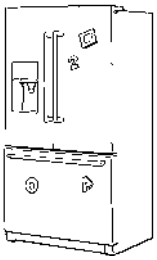
```
List<string> values = new List<string>();

values.Add("a value");

foreach (string value in values)
{
    values.Add("another value");
}
```

Exception Unhandled

System.InvalidOperationException: 'Collection was modified; enumeration operation may not execute.'

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings



Code Magnets

Oops! We had an app that produced this output nicely laid out on our refrigerator using snippets of code on magnets—but someone slammed the door and they all fell off! Can you rearrange the magnets to make a working console app that will write the correct output to the console?

Output

```
zero
one
three
four
4.2
```

Rearrange these magnets here to make the code produce the right output.

```
string zilch = "zero";
string first =  "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
}
```

```
{
```

```
void PppPppL (List<string> a)
```

```
foreach (string element in a)
{
    Console.WriteLine(element);
}
```

```
List<string> a = new List<string>();
```

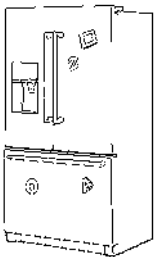
```
if (a.IndexOf("four") != 4)
{
        a.Add(fourth);

}
```

```
a.RemoveAt(2);
```

```
if (a.Contains("three"))
{
      a.Add("four");
}
```

```
PppPppL(a);
```

```
if (a.Contains("two")) {
   a.Add(twopointtwo);
}
```



Code Magnet Solution

Remember how we talked about using intuitive names back in Chapter 3? Well, that may make for good code, but it makes these puzzles a bit too easy. Just don't use cryptic names like PppPppL in real life!

Output

```
zero
one
three
four
4.2
```

```
List<string> a = new List<string>();

string zilch = "zero";
string first =  "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";

a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);

if (a.Contains("three"))
{
      a.Add("four");
}

a.RemoveAt(2);

if (a.IndexOf("four") != 4)
{
       a.Add(fourth);

}

if (a.Contains("two")) {
   a.Add(twopointtwo);
}

PppPppL(a);

void PppPppL (List<string> a)
{
    foreach (string element in a)
    {
        Console.WriteLine(element);
    }
}
```

Can you figure out why "2.2" never gets added to the list, even though it's declared here and passed to a.Add below? Use the debugger to sleuth it out!

RemoveAt removes the element at index #2—which is the third element in the list.

The PppPppL method uses a foreach loop to go through a list of strings, add each of them to one big string, and then show it in a message box.

The foreach loop goes through all of the elements in the list and prints them.

there are no Dumb Questions

Q: So why would I ever use an enum instead of a collection? Don't they kind of solve somewhat similar problems?

A: Enums do very different things from collections. First and foremost, enums are **types**, while collections are **objects**.

You can think of enums as a handy way to store ***lists of constants*** so you can refer to them by name. They're great for keeping your code readable and making sure that you are always using the right variable names to access values that you use really frequently.

A collection can store just about anything because it stores ***object references***, which can access the objects' members as usual. Enums, on the other hand, have to be assigned one of the value types in C# (like the ones introduced in Chapter 4). You can cast them to values, but not references.

Enums can't dynamically change their size either. They can't implement interfaces or have methods, and you'll have to cast them to another type to store a value from an enum in another variable. Add all of that up and you've got some pretty big differences between the two ways of storing data. Both are really useful in their own right.

Q: It sounds like the List class is pretty powerful. So why would I ever want to use an array?

A: If you need to store a collection of objects, you'll generally use a list and not an array. One place where you'll use arrays (which you'll see later in the book) is where you're reading sequences of bytes—for example, from a file. In that case, you'll often call a method on a .NET class that returns a byte[]. Luckily, it's easy to convert a list to an array (by calling its ToArray method), or an array to a List (using an overloaded list constructor).

Arrays actually take up less memory and CPU time for your programs, but that only accounts for a tiny performance boost. If you have to do the same thing, say, millions of times a second, you might want to use an array and not a list. But if your program is running slowly, it's pretty unlikely that switching from lists to arrays will fix the problem.

Q: I don't get the name "generic." Why is it called a generic collection?

A: A generic collection is a collection object (or a built-in object that lets you store and manage a bunch of other objects) that's been set up to store only one type (or more than one type, as you'll see shortly).

Q: OK, that explains the "collection" part. But what makes it "generic"?

A: Many years ago, supermarkets used to carry generic items that were packaged in white wrappers with black type that just said the name of what was inside ("Potato Chips," "Cola," "Soap," etc.). The generic brand was all about what was inside the bag, and not about how it was displayed.

The same thing happens with generic data types. Your List<T> will work exactly the same with whatever happens to be inside it. A list of Shoe objects, Card objects, ints, longs, or even other lists will still act at the container level. So you can always add, remove, insert, etc., no matter what's inside the list itself.

The term "generic" refers to the fact that even though a specific instance of List can only store one specific type, the List class in general works with any type. That's what the <T> is for—it says that the list contains a bunch of references of type T.

Q: Can I have a list that doesn't have a type?

A: No. Every list—in fact, every generic collection (and you'll learn about the other generic collections in just a minute)—must have a type connected to it. C# does have nongeneric lists called ArrayLists that can store any kind of object. If you want to use an ArrayList, you need to include a `using System.Collections;` line in your code. You should rarely need to do this, because a List<*object*> will usually work just fine in a scenario where you might want to use an untyped ArrayList.

When you create a new List object, you always supply a type—that tells C# what type of data it'll store. A List can store a value type (like int, bool, or decimal) or a class.

You can use collection expressions to create Lists

You've been using **collection expressions** to create arrays. They also work to create lists too. You can use a collection expression any time you need to create a new List object with an initial list of members. It works by creating the list and then adding all of the items to the list.

This code creates a new List<Shoe> and fills it with new Shoe objects by calling the Add method over and over again.

```
List<Shoe> shoeCloset = new List<Shoe>();
shoeCloset.Add(new Shoe(Style.Sneaker, "Black"));
shoeCloset.Add(new Shoe(Style.Clog, "Brown"));
shoeCloset.Add(new Shoe(Style.Wingtip, "Black"));
shoeCloset.Add(new Shoe(Style.Loafer, "White"));
shoeCloset.Add(new Shoe(Style.Loafer, "Red"));
shoeCloset.Add(new Shoe(Style.Sneaker, "Green"));
```

The same code rewritten using a collection expression

Your can create the same list with a collection expression with the same Shoe objects. It generates the same List, and the code is simpler and easier to read.

```
List<Shoe> shoeCloset = [
    new Shoe(Style.Sneaker, "Black"),
    new Shoe(Style.Clog, "Brown"),
    new Shoe(Style.Wingtip, "Black"),
    new Shoe(Style.Loafer, "White"),
    new Shoe(Style.Loafer, "Red"),
    new Shoe(Style.Sneaker, "Green"),
];
```

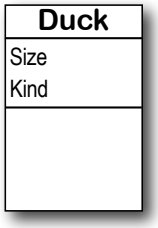
You're not limited to using "new" statements in the collection expression. You can include values and variables, just like you've been doing with arrays.

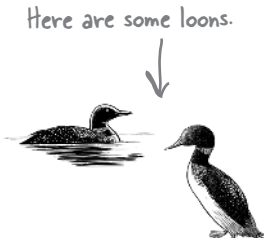
A collection expression makes your code simpler and easier to read by letting you combine creating a list with adding an initial set of items.

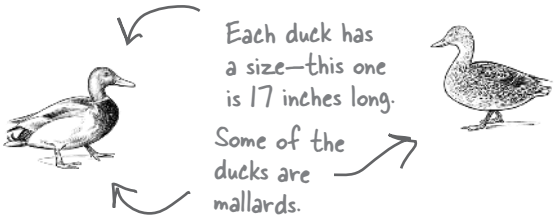
Let's create a List of Ducks

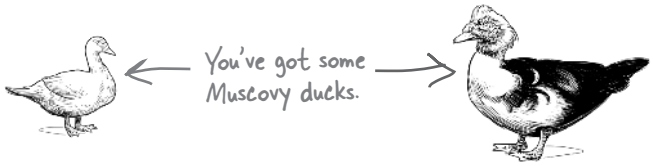
Do this!

Here's a Duck class that keeps track of your many neighborhood ducks. (You do collect ducks, don't you?) **Create a new Console App project** and add a new Duck class and KindOfDuck enum.









Here's the code for the Duck class and KindOfDuck enum. The Duck class has a primary constructor and two read-only properties to return the values of the private variables defined by the constructor:

```
class Duck(int size, KindOfDuck kind)
{
    public int Size { get { return size; } }
    public KindOfDuck Kind { get { return kind; } }
}
```

```
enum KindOfDuck
{
    Mallard,
    Muscovy,
    Loon,
}
```

Use a collection expression to create a List of Ducks

You've got six ducks, so you'll create a List<Duck> that has a collection expression with six new Duck objects. Each statement in it creates a new Duck, ***using its primary constructor*** to set each Duck object's Size and Kind fields. Add the Duck class and KindOfDuck enum to your project.

The KindOfDuck enum determines what sorts of ducks are in your collection. Notice that we're not assigning values—that's very typical. We won't need numerical values for ducks, so the default enum values (0, 1, 2...) will be just fine.

Here are the top-level statements for the app:

```
List<Duck> ducks = [
    new Duck(17, KindOfDuck.Mallard),
    new Duck(18, KindOfDuck.Muscovy),
    new Duck(14, KindOfDuck.Loon),
    new Duck(11, KindOfDuck.Muscovy),
    new Duck(14, KindOfDuck.Mallard),
    new Duck(13, KindOfDuck.Loon),
];

PrintDucks(ducks);

void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks) {
        Console.WriteLine($"{duck.Size} inch {duck.Kind}");
    }
}
```

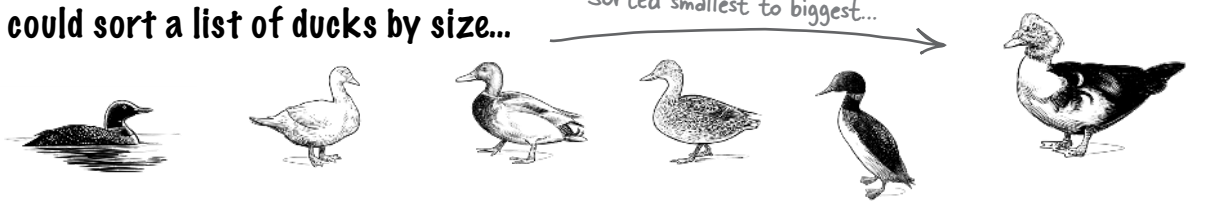
Now **run your code**. It will print each of the Ducks in your list to the console.

Sorting lists can be tricky

It's not hard to think about ways to sort numbers or letters. But how do you sort two individual objects, especially if they have multiple fields? In some cases, you might want to order objects by the value in the Name field, while in other cases it might make sense to order objects based on height or date of birth. There are lots of ways you can order things, and lists support all of them.

You could sort a list of ducks by size...

Sorted smallest to biggest...



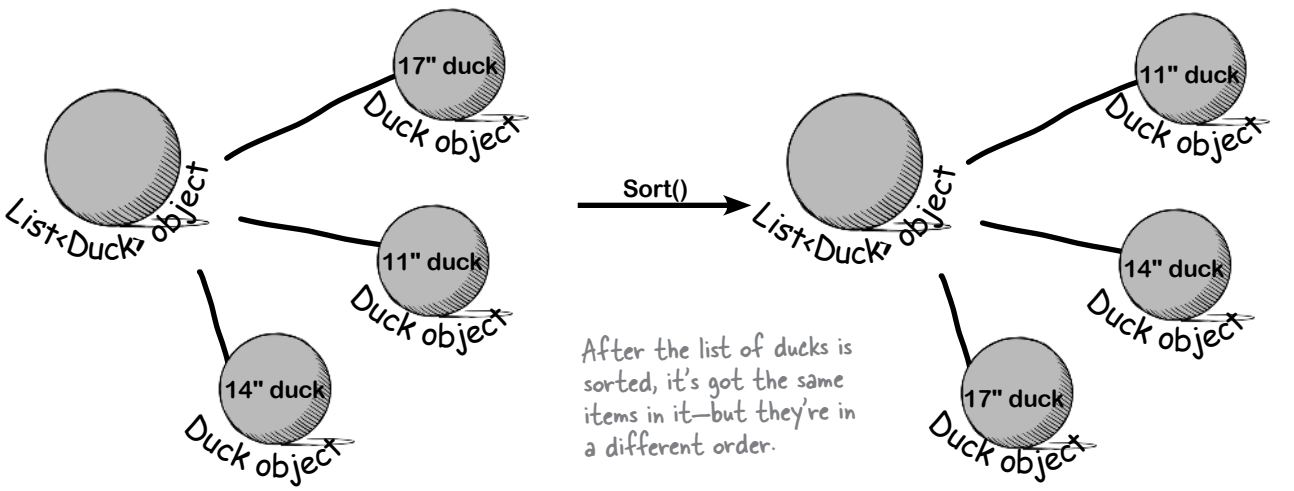
...or by kind.

Sorted by kind of duck...

Lists know how to sort themselves

Every List comes with a **Sort method** that rearranges all of the items in the list to put them in order. Lists already know how to sort most built-in types and classes, and it's easy to teach them how to sort your own classes.

Technically, it's not the List<T> that knows how to sort itself. This is the job of an IComparer<T> object, which you'll learn about in a minute.



After the list of ducks is sorted, it's got the same items in it—but they're in a different order.

IComparable<Duck> helps your List sort its Ducks

If you have a List of numbers and call its Sort method, it will sort the list with the smallest numbers first and largest last. How does the List know which way to sort the Duck objects? We tell List.Sort that the Duck class can be sorted—and we do that the way we typically indicate that a class can do a certain job, *with an interface*.

The List.Sort method knows how to sort any type or class that **implements the IComparable<T> interface**. That interface has just one member—a method called CompareTo. Sort uses an object's CompareTo method to compare it with other objects, and uses its return value (an int) to determine which comes first.

You can make any class work with List's built-in Sort method by having it implement IComparable<T> and adding a CompareTo method.

An object's CompareTo method compares it to another object

One way to give our List object the ability to sort ducks is to **modify the Duck class to implement IComparable<Duck>** and add its only member, a CompareTo method that takes a Duck reference as a parameter.

Update your project's Duck class by implementing IComparable<Duck> so that it sorts itself based on duck size:

You specify the type being compared when you have the class implement the IComparable<T> interface.

```
class Duck (int size, KindOfDuck kind) : IComparable<Duck>
{
    public int Size { get { return size; } }
    public KindOfDuck Kind { get { return kind; } }

    public int CompareTo(Duck? other)
    {
        if (other == null) return 0;

        if (this.Size > other.Size)
            return 1;
        else if (this.Size < other.Size)
            return -1;
        else
            return 0;
    }
}
```

Most CompareTo methods look a lot like this. This method first compares the Size field against the other Duck's Size field. If this Duck is bigger, it returns 1. If it's smaller, it returns –1. And if they're the same size, it returns 0.

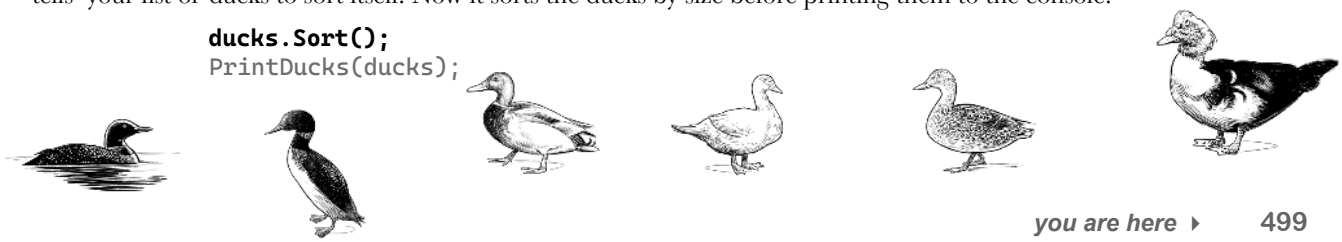
There's that ? again, like we saw with string?—it means other can be null. We'll learn more about that later in the book.

If you want to sort your list from smallest to biggest, make CompareTo return a positive number if it's comparing to a smaller duck, and a negative number if it's comparing to a bigger one.

If the duck to compare should come after the current duck in the sorted list, CompareTo needs to return a *positive* number. If it should come before, CompareTo returns a *negative* number. If they're the same, it returns *zero*.

Add this line of code to the end of your top-level statements just before the call to PrintDucks. This tells your list of ducks to sort itself. Now it sorts the ducks by size before printing them to the console:

```
ducks.Sort();
PrintDucks(ducks);
```



Use IComparer to tell your List how to sort

Your Duck class implements IComparable, so List.Sort knows how to sort a List of Duck objects. But what if you want to sort them in a different way than usual? Or what if you want to sort a type of object that doesn't implement IComparable? Then you can pass a **comparer object** as an argument to List.Sort, to give it a different way to sort its objects. Notice how List.Sort is overloaded:

```
▲ 3 of 4 ▼ void List<Duck>.Sort(IComparer<Duck>? comparer)
           Sorts the elements in the entire List<T> using the specified comparer.
           comparer: The IComparer<in T> implementation to use when comparing elements, or null to use the default comparer Comparer<T>.Default.
```

There's an overloaded version of List.Sort that **takes an IComparer<T> reference**, where T will be replaced by the generic type for your list (so for a List<Duck> it takes an IComparer<Duck> argument, for a List<string> it's an IComparer<string>, etc.). You'll pass it a reference to an object that implements an interface, and we know what that means: that it *does a specific job*. In this case, that job is comparing pairs of items in the list to tell List.Sort what order to sort them in.

The IComparer<T> interface has one member, a **method called Compare**. It's just like the CompareTo method in IComparable<T>: it takes two object parameters, `x` and `y`, and returns a positive value if `x` comes before `y`, a negative value if `x` comes after `y`, or zero if they're the same.

Add an IComparer to your project

Add the DuckComparerBySize class to your project. It's a comparer object that you can pass as a parameter to List.Sort to make it sort your ducks by size.

Here's the code for the comparer class:

```
class DuckComparerBySize : IComparer<Duck>
{
    public int Compare(Duck? x, Duck? y)
    {
        if (x == null || y == null) return 0;

        if (x.Size < y.Size)
            return -1;
        if (x.Size > y.Size)
            return 1;
        return 0;
    }
}
```

Either Duck can be null, and in that case consider them equal. (Can you think of a better way to handle that case?)

If Compare returns a negative number, that means object x should go before object y. x is "less than" y.

Any positive value means object x should go after object y. x is "greater than" y. Zero means they're "equal."

A comparer object is an instance of a class that implements IComparer<T> that you can pass as a reference to List.Sort. Its Compare method works just like the CompareTo method in the IComparable<T> interface. When List.Sort compares its elements to sort them, it passes pairs of objects to the Compare method in your comparer object, so your List will sort differently depending on how you implement the comparer.

Can you figure out how to modify DuckComparerBySize so it sorts the ducks largest to smallest instead?

Create an instance of your comparer object

When you want to sort using IComparer<T>, you need to create a new instance of the class that implements it—in this case, DuckComparerBySize. That's the comparer object that will help List.Sort figure out how to sort its elements. Like any other (nonstatic) class, you need to instantiate it before you use it:

Replace ducks.Sort in your top-level statements with these two lines of code. It still sorts the ducks, but now it uses the comparer object.

```
IComparer<Duck> sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

You'll pass Sort, a reference to the new DuckComparerBySize object as its parameter.

Multiple IComparer implementations, multiple ways to sort your objects

You can create multiple IComparer<Duck> classes with different sorting logic to sort the ducks in different ways. Then you can use the comparer you want when you need to sort in that particular way. Here's another duck comparer implementation to add to your project:

This comparer sorts by duck type. Remember, when you compare the enum Kind, you're comparing their enum index values. We didn't assign values when we declared the KindOfDuck enum, so they're given values 0, 1, 2, etc. in the order they appear in the enum declaration (so Mallard is 0, Muscovy is 1, and Loon is 2).

```
class DuckComparerByKind : IComparer<Duck> {
   public int Compare(Duck? x, Duck? y) {
       if (x == null || y == null) return 0;

       if (x.Kind < y.Kind)
           return -1;
       if (x.Kind > y.Kind)
           return 1;
       else
           return 0;
        }
}
```

We compared the ducks' Kind properties, so the ducks are sorted based on the index value of the Kind property, a KindOfDuck enum.

Notice how "greater than" and "less than" have different meanings here. We used < and > to compare enum index values, which lets us put the ducks in order.

Here's an example of how enums and lists work together. Enums stand in for numbers, and can be compared to sort lists.

Go back and modify your app to use this new comparer. Now it sorts the ducks by kind before it prints them.

```
IComparer<Duck> kindComparer = new DuckComparerByKind();
ducks.Sort(kindComparer);
PrintDucks(ducks);
```

Comparers can do complex comparisons

One advantage to creating a separate class for sorting your ducks is that you can build more complex logic into that class—and you can add members that help determine how the list gets sorted. Here's a more complex class to compare ducks. Its Compare method takes the same parameters, but it looks at the public SortBy field to determine how to sort the ducks.

```
enum SortCriteria {
    SizeThenKind,
    KindThenSize,
}

class DuckComparer : IComparer<Duck> {
    public SortCriteria SortBy = SortCriteria.SizeThenKind;

    public int Compare(Duck? x, Duck? y) {
        if (x == null || y == null) return 0;

        if (SortBy == SortCriteria.SizeThenKind)
            if (x.Size > y.Size)
                return 1;
            else if (x.Size < y.Size)
                return -1;
            else
                if (x.Kind > y.Kind)
                    return 1;
                else if (x.Kind < y.Kind)
                    return -1;
                else
                    return 0;
        else
            if (x.Kind > y.Kind)
                return 1;
            else if (x.Kind < y.Kind)
                return -1;
            else
                if (x.Size > y.Size)
                    return 1;
                else if (x.Size < y.Size)
                    return -1;
                else
                    return 0;
    }
}
```

This enum tells the comparer object which way to sort the ducks.

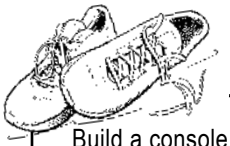
This "if" statement checks the SortBy field. If it's set to SizeThenKind, then it first sorts the ducks by size, and then within each size it'll sort the ducks by their kind.

Instead of just returning 0 if the two ducks are the same size, the comparer checks their kind, and only returns 0 if the two ducks are both the same size and the same kind.

If SortBy isn't set to SizeThenKind, then the comparer first sorts by the kind of duck. If the two ducks are the same kind, then it compares their size.

Add the DuckComparer class to your project, then add this code to the end of your top-level statements. It uses the comparer object, setting its SortBy field before calling ducks.Sort. Now you can change the way the list sorts its ducks by changing a property in the comparer.

```
DuckComparer comparer = new DuckComparer();
Console.WriteLine("\nSorting by kind then size\n");
comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);
Console.WriteLine("\nSorting by size then kind\n");
comparer.SortBy = SortCriteria.SizeThenKind;
ducks.Sort(comparer);
PrintDucks(ducks);
```



Exercise

Build a console app that creates a list of cards in random order, prints them to the console, uses a comparer object to sort the cards, and then prints the sorted list.

1 **Write a method to make a jumbled set of cards.**

Create a new console app. Add the Suits enum, Values enum, and Card class from earlier in the chapter. Then add two static methods to *Program.cs*: a RandomCard method that returns a reference to a card with a random suit and value, and a PrintCards method that prints a List<Card>.

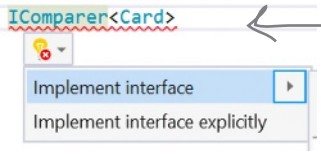
2 **Create a class that implements IComparer<Card> to sort the cards.**

Here's a good opportunity to use the IDE's **Quick Actions menu** to implement an interface. Add a class called CardComparerByValue, then make it implement the IComparer<Card> interface:

```
class CardComparerByValue : IComparer<Card>
```

Click on `IComparer<Card>` and hover over the `I`. You'll see a light bulb or screwdriver Quick Action icon appear.. When you click on the icon, the IDE pops up its Quick Actions menu:

The IDE has a useful shortcut that makes it easy to access the Quick Actions menu: press Ctrl+. (Windows) or Option+Enter (Mac).



Your IComparer object needs to sort the cards by value, so the cards with the lowest values are first in the list.

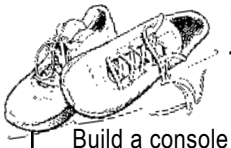
Choose "Implement interface" —that tells the IDE to automatically fill in all of the methods and properties in the interface that you need to implement. In this case, it creates an empty Compare method to compare two cards, `x` and `y`. Make it return 1 if `x` is bigger than `y`, –1 if it's smaller, and 0 if they're the same. First, order by suit: Diamonds come first, then Clubs, then Hearts, then Spades. Make sure that any King comes after any Jack, which comes after any 4, which comes after any Ace. You can compare enum values without casting: if (x.Suit < y.Suit).

3 **Make sure the output looks right.**

Write the top-level statements so the output looks like this.

- It prompts for a number of cards.
- If the user enters a valid number and presses Enter, it generates a list of random cards and then prints them.
- It sorts the list of cards using the comparer.
- It prints the sorted list of cards.

```
Enter number of cards: 9
Eight of Spades
Nine of Hearts
Four of Hearts
Nine of Hearts
King of Diamonds
King of Spades
Six of Spades
Seven of Clubs
Seven of Clubs

... sorting the cards ...

King of Diamonds
Seven of Clubs
Seven of Clubs
Four of Hearts
Nine of Hearts
Nine of Hearts
Six of Spades
Eight of Spades
King of Spades
```



Exercise Solution

Build a console app that creates a list of cards in random order, prints them to the console, uses a comparer object to sort the cards, and then prints the sorted list. Don't forget to add `using System.Collections.Generic;` to your *Program.cs* file and class file if they don't already have it. Here are the top-level statements:

```
List<Card> cards = new List<Card>();
Console.Write("Enter number of cards: ");
if (int.TryParse(Console.ReadLine(), out int numberOfCards))
    for (int i = 0; i < numberOfCards; i++)
        cards.Add(RandomCard());

PrintCards(cards);

cards.Sort(new CardComparerByValue());
Console.WriteLine("\n... sorting the cards ...\n");
PrintCards(cards);

void PrintCards(List<Card> cards)
{
    foreach (Card card in cards)
    {
        Console.WriteLine(card.Name);
    }
}

Card RandomCard()
{
    return new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4));
}
```

Here's where we create a generic List of Card objects to store the cards. Once they're in the list, it's easy to sort them using an IComparer.

We left out the curly brackets here. Do you think it makes the code easier or harder to read?

The cards are sorted using the comparer class.

Here's the CardComparerByValue class used to sort the cards:

```
class CardComparerByValue : IComparer<Card>
{
    public int Compare(Card? x, Card? y)
    {
        if (x == null || y == null) return 0;
        if (x.Suit < y.Suit)
            return -1;
        if (x.Suit > y.Suit)
            return 1;
        if (x.Value < y.Value)
            return -1;
        if (x.Value > y.Value)
            return 1;
        return 0;
    }
}
```

Here's the "guts" of the card sorting, which uses the built-in List.Sort method. Sort takes an IComparer object, which has one method: Compare. This implementation takes two cards and first compares their suits, then their values.

We want all Diamonds to come before all Clubs, so we need to compare the suits first. We can take advantage of the enum values.

These statements only get executed if x and y have the same value—that means the first two return statements weren't executed.

If none of the other five return statements were hit, the cards must be the same—so return zero.

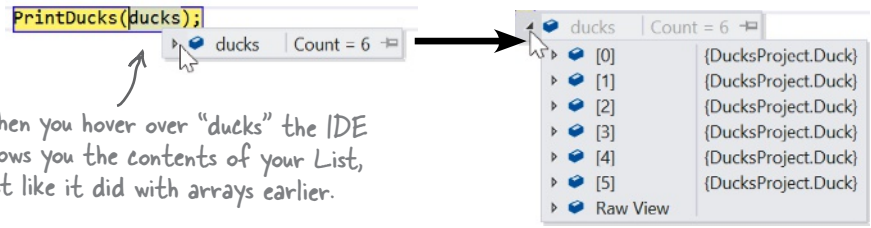
Overriding a ToString method lets an object describe itself

Every object has a **method called ToString that converts it to a string**. You've already used it—any time you use `{curly braces}` in string interpolation, that calls the ToString method of whatever's inside them—and the IDE also takes advantage of it. When you create a class, it inherits the ToString method from Object, the top-level base class that all other classes extend.

The Object.ToString method prints the **fully qualified class name**, or the namespace followed by a period followed by the class name. Since we used the namespace DucksProject when we were writing this chapter, the fully qualified class name for our Duck class is DucksProject.Duck:

```
Console.WriteLine(new Duck(14, KindOfDuck.Loon).ToString());  →  "DucksProject.Duck"
```

The IDE also calls the ToString method—for example, when you watch or inspect a variable:



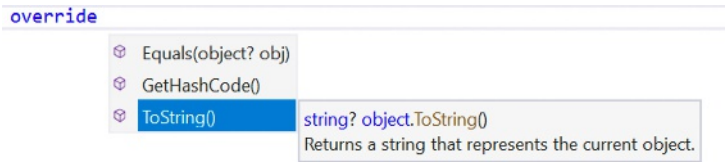
When you hover over "ducks" the IDE shows you the contents of your List, just like it did with arrays earlier.

The IDE calls the ToString method when you watch or inspect a variable, but the ToString method that Duck inherited from Object just returns its class name, which isn't all that useful here.

Hmm, that's not as useful as we'd hoped. You can see that there are six Duck objects in the list. If you expand a Duck, you can see its Kind and Size values. Wouldn't it be easier if you could see all of them at once?

Override the ToString method to see your Ducks in the IDE

Luckily, ToString is a virtual method of Object, the base class of every object. So all you need to do is **override the ToString method**—and when you do, you'll see the results immediately in the IDE's Watch window! Open up your Duck class and start adding a new method by typing **`override`**. As soon as you add a space, the IDE will show you the methods you can override:

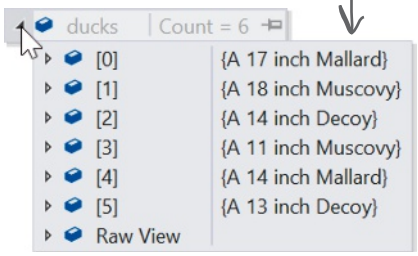


When the IDE's debugger shows you an object, it calls the object's ToString method.

Click on ToString() to tell the IDE to add a new ToString method. Replace the contents so it looks like this:

```
public override string ToString()
{
    return $"A {Size} inch {Kind}";
}
```

Run your program and look at the list again. Now the IDE shows you the contents of your Duck objects.



Update your foreach loops to let your Ducks and Cards write themselves to the console

You've seen two different examples of programs looping through a list of objects and calling Console.WriteLine to print a line to the console for each object—like this **foreach** loop that prints every Card in a List<Card>:

```
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
```

The PrintDucks method did something similar for Duck objects in a List:

```
foreach (Duck duck in ducks) {
    Console.WriteLine($"{duck.Size} inch {duck.Kind}");
}
```

This is a pretty common thing to do with objects. Now that your Duck has a ToString method, your PrintDucks method should take advantage of it. Use the IDE's IntelliSense to look through the overloads for the Console.WriteLine method—specifically this one:

▲ 10 of 18 ▼ void Console.WriteLine(**object value**)

If you pass Console.WriteLine a reference to an object, it will call that object's ToString method automatically.

You can pass any object to Console.WriteLine, and it will call its ToString method. So you can replace the PrintDucks method with one that calls this overload:

```
void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
}
```

Replace the PrintDucks method with this one and run your code again. It prints the same output. If you want to add, say, a Color or Weight property to your Duck object, you just have to update the ToString method, and everything that uses it (including the PrintDucks method) will reflect that change.

Add a ToString method to your Card object too

Your Card object already has a Name property that returns the name of the card:

```
public string Name {
    get { return $"{Value} of {Suit}"; }
}
```

That's exactly what its ToString method should do. So, add a ToString method to the Card class:

```
public override string ToString()
{
    return Name;
}
```

We decided to make the ToString method call the Name property. Do you think we made the right choice? Would it have been better to delete the Name property and move its code to the ToString method? When you're going back to modify your code, you have to make choices like this—and it's not always obvious which choice is best.

Now your programs that use Card objects will be easier to debug.

Sharpen your pencil

This app writes six lines to the console. Can you figure out what they are and write them down here?

```
List<Dog> dogs = [
    new Dog(Breeds.Dachshund, "Franz"),
    new Dog(Breeds.Collie, "Petunia"),
    new Dog(Breeds.Mastiff, "Pepperoni"),
    new Dog(Breeds.Dachshund, "Brunhilda"),
    new Dog(Breeds.Malanois, "Zippy"),
    new Dog(Breeds.Collie, "Carrie"),
];

dogs.Sort();
foreach (Dog dog in dogs)
    Console.WriteLine(dog);

enum Breeds
{
    Collie = 3,
    Malanois = -9,
    Dachshund = 7,
    Mastiff = 0,
}

class Dog(Breeds breed, string name) : IComparable<Dog>
{
    public Breeds Breed { get { return breed; } }

    public string Name { get { return name; } }

    public int CompareTo(Dog? other)
    {
        if (other == null) return 0;
        if (Breed > other.Breed) return -1;
        if (Breed < other.Breed) return 1;
        return -Name.CompareTo(other.Name);
    }

    public override string ToString()
    {
        return $"A {Breed} named {Name}";
    }
}
```

The List of Dog objects is created using a collection expression.

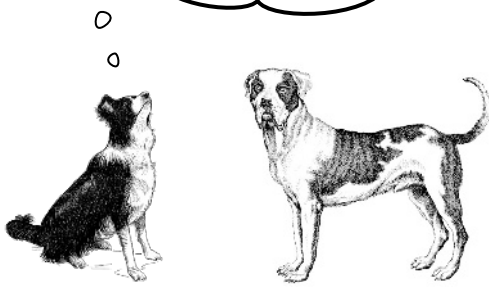
Hint—pay attention to the minus signs!

Here's a primary constructor and two read-only properties to return the values of the parameters.

The Dog class implements IComparable<Dog>—this is the CompareTo method it uses to implement that interface.

Can you figure out which one of the top-level statements calls each Dog object's ToString method?

See if you can figure out what this app prints to the console ***just from reading the code****, and without typing it in and running it.*



.....

.....

.....

.....

.....

.....

Sharpen your pencil Solution

Here's the output of the app. Did you get it right? It's OK if you didn't! Go back and take another look at the enum.

- Did you notice that the enumerators had different values?
- The Name property is a string, and strings also implement IComparable, so we can just call their CompareTo method to compare them.
- Also, take a closer look at the CompareTo method—did you notice that it returned 1 if the other breed was greater and –1 if the other breed was less, or the minus sign before Name.CompareTo(other.Name)? So first it's sorting by Breed and then by Name, but it's sorting both Breed and Name in reverse order.

When CompareTo uses > and < to compare Breed values, it uses the int values in the Breed enum declaration, so Collie is 3, Corgi is –9, Dachshund is 7, and Pug is 0.

Here's the output:

```
A Dachshund named Franz
A Dachshund named Brunhilda
A Collie named Petunia
A Collie named Carrie
A Mastiff named Pepperoni
A Malanois named Zippy
```

Bullet Points

- **Collection expressions** let you specify the contents of a List<T> just like you can do with arrays, with a comma-separated list of values inside [brackets].
- A **collection expression** makes your code more compact by letting you combine list creation with adding an initial set of items (but your code won't run more quickly).
- The **List.Sort method** sorts the contents of the collection, changing the order of the items it contains.
- The **IComparable<T> interface** contains a single method, CompareTo, which List.Sort uses to determine the order of objects to sort.
- An **overloaded method** is a method that you can call in more than one way, with different arrangements of parameters. The IDE's IntelliSense pop-up lets you scroll through the different overloads for a method.
- The **Sort method** has an overload that takes an IComparer<T> object, which it will then use for sorting.
- **IComparable.CompareTo** and **IComparer.Compare** both compare pairs of objects, returning –1 if the first object is less than the second, 1 if the first is greater than the second, or 0 if they're equal.
- The **String class implements IComparable**. An IComparer or IComparable for a class that includes a string member can call its Compare or CompareTo method to help determine the sort order.
- Every object has a **ToString method** that converts it to a string. The ToString method is called any time you use string interpolation or concatenation.
- The **default ToString** method is inherited from Object. It returns the fully qualified class name, or the namespace followed by a period followed by the class name.
- **Override the ToString method** in a class to get it to return a custom string any time it's used with string interpolation, concatenation, and many other operations.

foreach Loops Up Close

Let's take a closer look at `foreach` loops. Go to the IDE, find a List<Duck> variable, and use IntelliSense to take a look at its GetEnumerator method. Start typing **`.GetEnumerator`** and see what comes up:

```
ducks.GetEnumerator();
```

List<Duck>.Enumerator List<Duck>.GetEnumerator()
Returns an enumerator that iterates through the List<T>.

Returns:
A List<T>.Enumerator for the List<T>.

Create an Array[Duck] and do the same thing—the array also has a GetEnumerator method. That's because lists, arrays, and other collections implement an interface called **IEnumerable<T>**.

You already know that interfaces are all about making different objects do the same job. When an object implements the IEnumerable<T> interface, the specific job it does is that ***it supports iteration over a nongeneric collection***—in other words, it lets you write code that loops through it. Specifically, that means you can use it with a `foreach` loop.

So what does that look like under the hood? Use Go To Definition/Declaration on `List<Duck>` to see the interfaces that it implements, just like you did earlier. Then do it again to see the members of IEnumerable<T>. What do you see?

The IEnumerable<T> interface contains a single member: a method called GetEnumerator, which returns an **Enumerator object**. The Enumerator object provides the machinery that lets you loop through a list in order. So when you write this `foreach` loop:

```
foreach (Duck duck in ducks) {
    Console.WriteLine(duck);
}
```

Here's what that loop is actually doing behind the scenes:

```
IEnumerator<Duck> enumerator = ducks.GetEnumerator();
while (enumerator.MoveNext()) {
    Duck duck = enumerator.Current;
    Console.WriteLine(duck);
}
if (enumerator is IDisposable disposable) disposable.Dispose();
```

Technically, there's a little more code than this, but this is enough to give you the basic idea of what's going on.

Both loops write the same Ducks to the console. See this for yourself by running both of them—they'll both have the same output. (And don't worry about the last line for now; you'll learn about IDisposable in Chapter 10.)

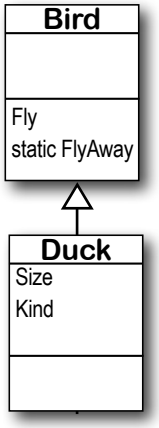
When you're looping through a list or array (or any other collection), the MoveNext method returns true if there's another element in the collection, or false if the enumerator has reached the end. The Current property always returns a reference to the current element. Add it all together, and you get a `foreach` loop.

When a collection implements IEnumerable<T>, that gives you a way to write a loop that goes through its contents in order.

You can upcast an entire list using IEnumerable<T>

Remember how you can upcast any object to its superclass? Well, when you've got a List of objects, you can upcast the entire list at once. It's called **covariance**, and all you need for it is an IEnumerable<T> interface reference.

Let's see how this works. We'll start with the Duck class that you've been working with throughout the chapter. Then we'll add a Bird class that it will extend. The Bird class will include a static method that iterates over a collection of Bird objects. Can we get it to work with a List of Ducks?



Here's the Bird class that you'll make your Duck class extend. You'll change its declaration to extend Bird, but leave the rest of the class the same. Then you'll add them both to a console app so you can experiment with covariance.

Do this!

Since all Ducks are Birds, covariance lets us convert a collection of Ducks to a collection of Birds. That can be really useful if you have to pass a List<Duck> to a method that only accepts a List<Bird>.

1. **Create a new Console App project.** Add a base class, Bird (for Duck to extend), and a Penguin class. We'll use the ToString method to make it easy to see which class is which.

```
class Bird
{
    public virtual void Fly(string destination)
    {
        Console.WriteLine($"{this} is flying to {destination}");
    }

    public override string ToString()
    {
        return $"A bird";
    }

    public static void FlyAway(List<Bird> flock, string destination)
    {
        foreach (Bird bird in flock)
        {
            bird.Fly(destination);
        }
    }
}
```

Covariance is C#'s way of letting you *implicitly convert* a subclass reference to its superclass. That word "implicitly" just means C# can figure out how to do the conversion without needing to explicitly use casting.

The static FlyAway method works with a collection of Birds. But what if we want to pass a List of Ducks to it?

2. **Add your original Duck class to the application.** Modify its declaration to **make it extend Bird**. You'll also need to **add the KindOfDuck enum** from earlier in the chapter:

```
class Duck (int size, KindOfDuck kind) : Bird {
    public int Size { get { return size; } }
    public KindOfDuck Kind { get { return kind; } }

    public override string ToString()
    {
        return $"A {Size} inch {Kind}";
    }
}
```

Add : Bird to the declaration to make the Duck class extend Bird. The rest of the Duck class is exactly the same.

```
enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

The Duck.Kind property uses KindOfDuck, so you'll need to add it too.

3. **Create the List<Duck> collection.** Go ahead and **add this code to your top-level statements**. It's the code from earlier in the chapter, plus one line to upcast it to a List<Bird>:

```
List<Duck> ducks = [
    new Duck(17, KindOfDuck.Mallard),
    new Duck(18, KindOfDuck.Muscovy),
    new Duck(14, KindOfDuck.Loon),
    new Duck(11, KindOfDuck.Muscovy),
    new Duck(14, KindOfDuck.Mallard),
    new Duck(13, KindOfDuck.Loon),
];

Bird.FlyAway(ducks, "Minnesota");
```

Copy the same collection expression you've been using to initialize your List of Ducks.

CS1503	Argument 1: cannot convert from 'System.Collections.Generic.List<Duck>' to 'System.Collections.Generic.List<Bird>'

Uh-oh—that code won't compile. The error message is telling you that you can't convert your Duck collection to a Bird collection. Let's try assigning `ducks` to a List<Bird>:

```
List<Bird> upcastDucks = ducks;
```

Well, that didn't work. We got a different error, but it still says we can't convert the type:

CS0029	Cannot implicitly convert type 'System.Collections.Generic.List<Duck>' to 'System.Collections.Generic.List<Bird>'

That error makes sense—it's similar to safely upcasting versus downcasting, which you learned about in Chapter 6: we can use assignment to downcast, but we need to use the `is` keyword to safely upcast. So how do we safely upcast our collection of Duck references to a more abstract collection of Bird references?

4. **Use covariance to make your ducks fly away.** That's where **covariance** comes in: you can ***use assignment to upcast your List<Duck> to an IEnumerable<Bird>***. Once you've got your IEnumerable<Bird>, you can call its ToList method to convert it to a List<Bird>:

```
IEnumerable<Bird> upcastDucks = ducks;
Bird.FlyAway(upcastDucks.ToList(), "Minnesota");
```

Now your collection of Duck references has been converted to a collection of Bird references. Fly away, little ducks!

Use a Dictionary to store keys and values

A list is like a big, long page full of names. What if you also want, for each name, an address? Or for every car in the `garage` list, you want details about that car? You need another kind of .NET collection: a **Dictionary**. A Dictionary lets you take a special value—the **key**—and associate that key with a bunch of data—the **value**. One more thing: a specific key can ***only appear once*** in any dictionary.

In a real-world dictionary, the word being defined is the key. You use it to look up the value, or the definition of the word.

dic•tion•ar•y, noun.

a book that lists the words of a language in alphabetical order and gives their meaning.

The definition is the value. It's the data associated with a particular key (in this case, the word being defined).

Here's how you declare a .NET Dictionary in C#:

```
Dictionary<TKey, TValue> dict = new Dictionary<TKey, TValue>();
```

These are the generic types for the Dictionary. TKey is the type used for the key that looks up the values, and TValue is the type of the values. So if you're storing words and their definitions, you'd use a Dictionary<string, string>. If you wanted to keep track of the number of times each word appears in a book, you could use Dictionary<string, int>.

These represent types. The first type in the angle brackets is always the key, and the second is always the data.

How to use a Dictionary

You can create a Dictionary that stores strings and uses strings as keys like this:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
```

You can store or look up a value in that Dictionary like this: `myDictionary["key"]`

Let's see a Dictionary in action. Here's a small console app that uses a `Dictionary<string, string>` to keep track of the favorite foods of a few friends:

```
Dictionary<string, string> favoriteFoods = new Dictionary<string, string>();
favoriteFoods["Alex"] = "hot dogs";
favoriteFoods["A'ja"] = "pizza";
favoriteFoods["Jules"] = "falafel";
favoriteFoods["Naima"] = "spaghetti";

string? name;
while (!String.IsNullOrEmpty(name = Console.ReadLine()))
{
    if (name != null && favoriteFoods.ContainsKey(name))
        Console.WriteLine($"{name}'s favorite food is {favoriteFoods[name]}");
    else
        Console.WriteLine($"I don't know {name}'s favorite food");
}
```

We're adding four key/value pairs to our Dictionary. In this case, the key is a person's name, and the value is that person's favorite food.

A Dictionary's ContainsKey method returns true if it contains a value for a specific key.

Here's how you get the value for a key.

The Dictionary functionality rundown

Dictionaries are a lot like lists. Both types are flexible in letting you work with lots of data types and come with lots of built-in functionality. Here are the basic things you can do with a Dictionary.

★ Add an item.

You can add an item to a Dictionary using its **indexer** with square brackets:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary["some key"] = "some value";
```

You can also add an item to a dictionary using its **Add method**:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary.Add("some key", "some value");
```

★ Look up a value using its key.

The most important thing you'll do with a Dictionary is **look up values with the indexer**—which makes sense, because you stored those values in a Dictionary so you could look them up using their unique keys. This example shows a Dictionary<string, string>, so we'll look up values using a string key, and the Dictionary returns a string value:

```
string lookupValue = myDictionary["some key"];
```

★ Remove an item.

Just like with a list, you can remove an item from a Dictionary using the **Remove method**. All you need to pass to the Remove method is the key value to have both the key and the value removed:

```
myDictionary.Remove("some key");
```

Keys are unique in a Dictionary; any key appears exactly once. Values can appear any number of times—two keys can have the same value. That way, when you look up or remove a key, the Dictionary knows what to remove.

★ Get a list of keys.

You can get a list of all of the keys in a Dictionary using its **Keys property** and loop through it using a `foreach` loop. Here's what that would look like:

```
foreach (string key in myDictionary.Keys) { ... };
```

Keys is a property of your Dictionary object. This particular Dictionary has string keys, so Keys is a collection of strings.

★ Count the pairs in the dictionary.

The **Count property** returns the number of key/value pairs that are in the Dictionary:

```
int howMany = myDictionary.Count;
```

Your key and value can be different types

It's common to see a Dictionary that maps integers to objects when you're assigning unique ID numbers to objects.

Dictionaries are versatile! They can hold just about anything, not just value types but ***any kind of object***. Here's an example of a Dictionary with a string as a key and a Duck object reference as a value:

```
Dictionary<string, Duck> duckIds = new Dictionary<int, Duck>();
duckIds.Add("Edith", new Duck(15, KindOfDuck.Mallard);
```

Collection Expressions Up Close

You've been using collection expressions since the first chapter of the book, where you used one to create a List of emoji characters for your animal matching game (go back and reread that code now!). When you create a collection expression, it's automatically converted to **many different collection types**.

Let's experiment with collection expressions. Start the debugger and add a watch for `[1, 2, 3, 4, 5]`:

Name	Value
⊗ [1, 2, 3, 4, 5]	error CS9176: There is no target type for the collection expression.

You got an error because every collection expression **must have a type to convert to**. The C# compiler needs a way to recognize the type of collection to convert the expression to.

In Chapter 4, you used a collection expression to create an array of breads for Sloppy Joe:

```
string[] Breads = [ "rye", "white", "wheat", "pumpernickel", "a roll" ];
```

The C# compiler saw that Breads is a string array, so it converted the collection expression to a `string[]` value.

You can use a collection expression for a List—in fact, you can use it with any class that has an Add method:

```
List<int> oneToFive = [1, 2, 3, 4, 5];
```

The compiler detects that oneToFive is a type that *implements* `IEnumerable<T>` *and has an Add method*, and calls its Add method for each of the items in the collection.

You can also assign a collection expression to a variable with an IEnumerable type:

```
IEnumerable<int> sixToTen = [6, 7, 8, 9, 10];
```

In this case, it creates a new array (specifically, a type called ReadOnlyArray) with the values.

A collection expression can include a **spread element** by putting `..` in front of the name of another collection:

```
IEnumerable<int> numbers = [-1, 0, ..oneToFive, ..sixToTen, 11, 12];
```

This causes it to include all of the elements from the other collection—in this case, the final collection would contain these values: `[-1, -0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12].`

Need to initialize a Dictionary? Use a collection initializer.

If you want to create a new Dictionary with an initial set of items, you can call its Add method a bunch of times. You can also use a **collection initializer**—that's an older feature of C# that predates collection expressions. It starts with a `new` statement followed by an opening bracket, a comma-separated list of values to add, and a closing bracket. Here's an example of a collection initializer for a Dictionary with AI chatbots and their URLs:

```
Dictionary<string, string> aiChatbots = new Dictionary<string, string>() {
    { "ChatGPT", "https://chat.openai.com" },
    { "Copilot", "https://copilot.microsoft.com" },
    { "Bard", "https://bard.google.com" },
};
```

Collection expressions work with any class that extends IEnumerable and has an Add method that takes one argument. The Dictionary.Add method takes two parameters, the `key` and the `value`, so you need to use a collection initializer.

Write an app that uses a Dictionary

Do this!

Here's a quick app that New York Yankees baseball fans will like. When an important player retires, the team retires the player's jersey number. **Create a new console app** that looks up some Yankees who wore famous numbers and when those numbers were retired. Here's a class to keep track of a retired baseball player:

```
class RetiredPlayer
{
    public string Name { get; private set; }
    public int YearRetired { get; private set; }

    public RetiredPlayer(string player, int yearRetired)
    {
        Name = player;
        YearRetired = yearRetired;
    }
}
```

Yogi Berra was #8 for the New York Yankees, while Cal Ripken Jr. was #8 for the Baltimore Orioles. In a Dictionary, you can have duplicate values, but every key must be unique. Can you think of a way to store retired numbers for multiple teams?

The top-level statements in your app add retired players to a Dictionary. It uses jersey number as the Dictionary key because it's **unique**—once a jersey number is retired, the team ***never uses it again***. That's important to consider when designing an app that uses a Dictionary: you never want to discover your key is not as unique as you thought!

Use a collection initializer to create the Dictionary

A **collection initializer** lets you initialize a collection—but unlike collection expressions, which require a collection type to have an Add method that takes only one parameter, you can use collection initializers with a Dictionary. To use a collection initializer, use the `new` keyword to instantiate the collection, then follow it with a `{` curly bracket, then a series of values (just like you specify in a collection expression), and a closing curly bracket `}`. You can use a collection initializer to initialize a List or Array. Here's how you **use a collection initializer to initialize a dictionary**:

```
new Dictionary<TKey, TValue>() {
    {Key1, Value1},
    {Key2, Value2},
    {Key3, Value3},
};
```

Add each element to the Dictionary by putting the key and value in { brackets }separated by a comma, with a comma after each element.

Add these **top-level statements for your app** to initialize a Dictionary and loop through it:

```
Dictionary<int, RetiredPlayer> retiredYankees = new Dictionary<int, RetiredPlayer>() {
    {3, new RetiredPlayer("Babe Ruth", 1948)},
    {4, new RetiredPlayer("Lou Gehrig", 1939)},
    {5, new RetiredPlayer("Joe DiMaggio", 1952)},
    {7, new RetiredPlayer("Mickey Mantle", 1969)},
    {8, new RetiredPlayer("Yogi Berra", 1972)},
    {10, new RetiredPlayer("Phil Rizzuto", 1985)},
    {23, new RetiredPlayer("Don Mattingly", 1997)},
    {42, new RetiredPlayer("Jackie Robinson", 1993)},
    {44, new RetiredPlayer("Reggie Jackson", 1993)},
};
```

Use a collection initializer to populate your Dictionary with keys and values.

```
foreach (int jerseyNumber in retiredYankees.Keys)
{
    RetiredPlayer player = retiredYankees[jerseyNumber];
    Console.WriteLine($"{player.Name} #{jerseyNumber} retired in {player.YearRetired}");
}
```

This foreach loop iterates through the keys and writes a line for each retired player in the collection.

And yet MORE collection types...

List and Dictionary are two of the most frequently used collection types that are part of .NET. Lists and dictionaries are very flexible—you can access any of the data in them in any order. But sometimes you're using a collection to represent a bunch of things in the real world that need to be accessed in a specific order. You can restrict how your code accesses the data in a collection by using **a Queue or a Stack**. Those are generic collections like List<T>, but they're especially good at making sure that your data is processed in a certain order.

There are other types of collections too—but these are the ones that you're most likely to come in contact with.

Use a Queue when the first object you store will be the first one you'll use, like with:

- Cars moving down a one-way street
- People standing in line
- Customers on hold for a customer service support line
- Anything else that's handled on a first-come, first-served basis

A queue is first in, first out, which means that the first object that you put into the queue is the first one you pull out of it to use.

Use a Stack when you always want to use the object you stored most recently, like with:

- Furniture loaded into the back of a moving truck
- A stack of books where you want to read the most recently added one first
- People boarding or leaving a plane
- A pyramid of cheerleaders, where the ones on top have to dismount first… imagine the mess if the one on the bottom walked away first!

A stack is last in, first out: the first object that goes into the stack is the last one that comes out of it.

Generic .NET collections implement IEnumerable

Almost every large project that you'll work on will include some sort of generic collection, because your programs need to store data. When you're dealing with groups of similar things in the real world, they almost always naturally fall into a category that corresponds pretty well to one of these kinds of collections. No matter which of these collection types you use—List, Dictionary, Stack, or Queue—you'll always be able to use a `foreach` loop with them because all of them implement IEnumerable<T>.

You can, however, use foreach to enumerate through a stack or queue, because they implement IEnumerable!

A queue is like a list that lets you add objects to the end and use the ones at the beginning. A stack only lets you access the last object you put into it.

A queue is FIFO—first in, first out

A **queue** is a lot like a list, except that you can't just add or remove items at any index. To add an object to a queue, you **enqueue** it. That adds the object to the end of the queue. You can **dequeue** the first object from the front of the queue. When you do that, the object is removed from the queue, and the rest of the objects in the queue move up a position.

After the first Dequeue call, the first item in the queue is removed and returned, and the second item shifts into the first place.

```
// Create a Queue and add four strings to it
Queue<string> myQueue = new Queue<string>();
myQueue.Enqueue("first in line");
myQueue.Enqueue("second in line");
myQueue.Enqueue("third in line");
myQueue.Enqueue("last in line");
```

Here's where we call Enqueue to add four items to the queue. When we pull them out of the queue, they'll come out in the same order they went in.

```
// Peek "looks" at the first item in the queue without removing it
Console.WriteLine($"Peek() returned:\n{myQueue.Peek()}");        ①

// Dequeue pulls the next item from the FRONT of the queue
Console.WriteLine(
    $"The first Dequeue() returned:\n{myQueue.Dequeue()}");      ②
Console.WriteLine(
    $"The second Dequeue() returned:\n{myQueue.Dequeue()}");     ③

// Clear removes all of the items from the queue
Console.WriteLine($"Count before Clear():\n{myQueue.Count}");    ④
myQueue.Clear();
Console.WriteLine($"Count after Clear():\n{myQueue.Count}");     ⑤
```

Objects in a queue need to wait their turn. The first one in the queue is the first one to come out of it.

Output

```
① Peek() returned:
  first in line
② The first Dequeue() returned:
  first in line
③ The second Dequeue() returned:
  second in line
④ Count before Clear():
  2
⑤ Count after Clear():
  0
```

A stack is LIFO—last in, first out

A **stack** is really similar to a queue—with one big difference. You **push** each item onto a stack, and when you want to take an item from the stack, you **pop** one off of it. When you pop an item off of a stack, you end up with the most recent item that you pushed onto it. It's just like a stack of plates, magazines, or anything else—you can drop something onto the top of the stack, but you need to take it off before you can get to whatever's underneath it.

```
// Create a Stack and add four strings to it
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
```

Creating a stack is just like creating any other generic collection.

When you push an item onto a stack, it pushes the other items back one slot and sits on top.

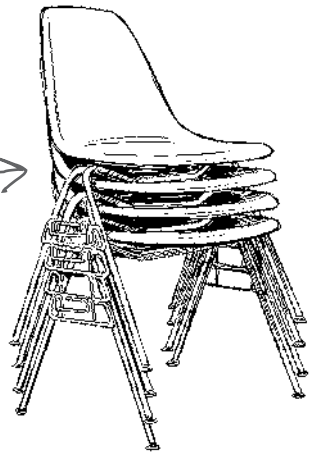
```
// Peek with a stack works just like it does with a queue
Console.WriteLine($"Peek() returned:\n{myStack.Peek()}");

// Pop pulls the next item from the TOP of the stack
Console.WriteLine($"The first Pop() returned:\n{myStack.Pop()}");
Console.WriteLine($"The second Pop() returned:\n{myStack.Pop()}");

// Clear removes all of the items from the stack
Console.WriteLine($"Count before Clear():\n{myStack.Count}");
myStack.Clear();
Console.WriteLine($"Count after Clear():\n{myStack.Count}");
```

1 When you pop an item off the stack, you get the most recent item that was added.

Output

```
1 Peek() returned:
  last in line
2 The first Pop() returned:
  last in line
3 The second Pop() returned:
  third in line
4 Count before Clear():
  2
5 Count after Clear():
  0
```

The last object you put on a stack is the first object that you pull off of it.



Wait a minute, something's bugging me. You haven't shown me anything I can do with a stack or a queue that I can't do with a list. They just save me a couple of lines of code, but I can't get at the items in the middle of a stack or a queue. I can do that with a list pretty easily! So why would I give that up just for ***a little convenience****?*

You don't give up anything when you use a queue or a stack.

It's really easy to copy a Queue object to a List object. It's just as easy to copy a List to a Queue, a Queue to a Stack…in fact, you can create a List, Queue, or Stack from any other object that implements the IEnumerable<T> interface. All you have to do is **use the overloaded constructor** that lets you pass the collection you want to copy from as a parameter. You can even **pass a collection expression to the constructor**. But remember, you're making a copy, which means you're creating a whole new object and adding it to the heap.

Try experimenting with creating copies of collections. Create a new console app, add this code, and use the debugger to step through it. Start by using a **collection initializer to create an array**. Then copy that array to a Queue, then copy the Queue to a List, then the List to a Stack.

Do this!

```
string[] myArray = new string[] {
    "first in line",
    "second in line",
    "third in line",
    "last in line",
};

Queue<string> myQueue = new Queue<string>(myArray);
List<string> myList = new List<string>(myQueue);
Stack<string> anotherStack = new Stack<string>(myList);

Console.WriteLine($@"myQueue has {myQueue.Count} items
myList has {myList.Count} items
anotherStack has {anotherStack.Count} items");
```

You can use a collection initializer to specify the initial elements of an array, List, Dictionary, or any class that *implements IEnumerable and has an Add method* (which, unfortunately, Stack and Queue do not have).

Copy that stack to a queue by creating a new queue and passing the stack to its constructor. Then copy the queue to a list, and then copy the list to another stack.

All four items were copied into the new collections.

Output

```
myQueue has 4 items
myList has 4 items
anotherStack has 4 items
```

…and you can always use a `foreach` loop to access all of the members in a stack or a queue!

Level up your prompt engineering for researching C# topics

This book doesn't have enough pages to cover every single topic related to C#, programming, or becoming a great developer. Great developers never stop learning new techniques, concepts, and tools, because technology keeps moving forward. That's why we've included many programming projects and code puzzles—we want to help you become a great learner, and the most effective way to get any development idea into your brain is to write lots of code.

An AI chatbot can be a **really great research tool** to help you learn. When you want to learn about a new C# topic, try these prompt engineering tactics (and read the OpenAI prompt engineering guide to learn about more of them!).

Find a persona that works right for you

In Chapter 7, we talked about adopting a persona—that's one of the tactics from the OpenAI prompt engineering guide. Small changes to the persona can make a big difference in how easily you learn from the answer. Some people learn better when they read answers in a more casual tone, others respond well to quick, straight-to-the-point answers. Try these prompts to see if one or the other works better for you:

```
Answer all of my questions in a light, casual tone. Choose whimsical
examples for any code samples that you generate.

Answer all of my questions in terse, precise language. Make all of the
code samples that you generate have to do with sci-fi spaceships.
```

Tell the chatbot how long you want the answer to be

Are you the sort of person who likes to have a really long and thorough answer? Or do you learn better with answers that are short and to the point? Try these prompts to see which style helps you absorb knowledge more effectively:

```
Give only short, precise answers in bullet point format that are no longer
than 6 lines long, and limit each bullet to 2 sentences maximum.

Provide long answers in full paragraphs of at least 6 sentences each.
```

Ask follow-up questions, ask the AI to reword its previous answer, and ask for lots of examples

When you start a new chat with an AI chatbot, it remembers everything it's told you so far. That means you can ask lots of follow-up questions. Is there something that wasn't clear? Ask it to explain the same thing. Ask for more detail. Ask for specific examples in a subject area that you understand and know very well—like using characters from your favorite anime—because that will help form lasting associations in your brain.

Try researching a new collection we haven't told you about: HashSet<T>

You've learned about List, Dictionary, Queue, and Stack—but those aren't the only C# collection types. There's another one that we haven't talked about: **HashSet**. Use an AI chatbot to answer these questions:

- What is a HashSet used for?
- Why would I use a HashSet instead of a List?
- Can you give examples of a code that needs a HashSet to solve a problem?

Take your time and really explore HashSets. Learning to research new topics is a valuable development skill.

Exercise

Write a program to help a cafeteria full of lumberjacks eat some flapjacks. You'll use a Queue of Lumberjack objects, and each Lumberjack has a Stack of Flapjack enums. We've given you some details as a starting point. Can you create a console app that matches the output?

Start with a Lumberjack class and a Flapjack enum

The Lumberjack class has a primary constructor with a Name argument, a read-only Name property to expose it, and a private Stack<Flapjack> flapjackStack field initialized with an empty stack.

The TakeFlapjack method takes a single argument, a Flapjack, and pushes it onto the stack. EatFlapjacks pops the flapjacks off the stack and writes the lines to the console for the lumberjack.

Lumberjack
Name private flapjackStack
TakeFlapjack EatFlapjacks

```
enum Flapjack {
    Crispy,
    Soggy,
    Browned,
    Banana,
}
```

Then add the top-level statements

The app prompts the user for the first lumberjack's name, then asks for the number of flapjacks to give it. If the user gives a valid number, the program calls TakeFlapjack that number of times, passing it a random Flapjack each time, and adds the Lumberjack to a Queue. It keeps asking for more Lumberjacks until the user enters a blank line, then it uses a `while` loop to dequeue each Lumberjack and calls its EatFlapjacks method to write lines to the output.

The top-level statements write these lines and take input, creating each Lumberjack object, setting its name, having it take a number of random flapjacks, and adding it to a queue.

```
Number of flapjacks: 4
Next lumberjack's name (blank to end): Hildur
Number of flapjacks: 6
Next lumberjack's name (blank to end): Jan
Number of flapjacks: 3
Next lumberjack's name (blank to end): Betty
Number of flapjacks: 4
Next lumberjack's name (blank to end):
Erik is eating flapjacks
Erik ate a soggy flapjack
Erik ate a crispy flapjack
Erik ate a crispy flapjack
Erik ate a soggy flapjack
Hildur is eating flapjacks
Hildur ate a browned flapjack
Hildur ate a browned flapjack
Hildur ate a banana flapjack
Hildur ate a crispy flapjack
Hildur ate a browned flapjack
Hildur ate a browned flapjack
Jan is eating flapjacks
Jan ate a soggy flapjack
Jan ate a browned flapjack
Jan ate a crispy flapjack
Betty is eating flapjacks
Betty ate a soggy flapjack
Betty ate a banana flapjack
Betty ate a browned flapjack
Betty ate a soggy flapjack
```

When the user is done entering lumberjacks, the top-level statements use a while loop to dequeue each Lumberjack and call its EatFlapjacks method. The rest of the output lines are written by each Lumberjack object.

Lumberjacks write this line when they start eating flapjacks.

This lumberjack took four flapjacks. When her EatFlapjacks method was called, she popped four Flapjack enums off of her stack.

Exercise Solution

Here's the code for the top-level statements:

```
Queue<Lumberjack> lumberjacks = new Queue<Lumberjack>();

string? name;
Console.Write("First lumberjack's name: ");
while (!String.IsNullOrEmpty(name = Console.ReadLine()))
{
    Console.Write("Number of flapjacks: ");
    if (int.TryParse(Console.ReadLine(), out int number))
    {
        Lumberjack lumberjack = new Lumberjack(name);
        for (int i = 0; i < number; i++)
        {
            lumberjack.TakeFlapjack((Flapjack)Random.Shared.Next(0, 4));
        }
        lumberjacks.Enqueue(lumberjack);
    }
    Console.Write("Next lumberjack's name (blank to end): ");
}

while (lumberjacks.Count > 0)
{
    Lumberjack next = lumberjacks.Dequeue();
    next.EatFlapjacks();
}
```

The app keeps its Lumberjack references in a queue.

It creates each Lumberjack object, calls its TakeFlapjack method with random flapjacks, and then enqueues the reference.

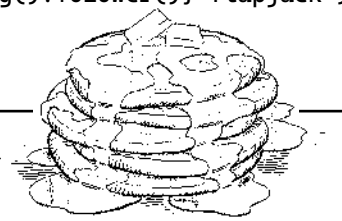
When the user is done adding flapjacks, the top-level statements use a while loop to dequeue each Lumberjack reference and call its EatFlapjacks method.

Here's the code for the Lumberjack class:

```
class Lumberjack(string name)
{
    public string Name { get { return name; } }

    private Stack<Flapjack> flapjackStack = new Stack<Flapjack>();

    public void TakeFlapjack(Flapjack flapjack)
    {
        flapjackStack.Push(flapjack);
    }

    public void EatFlapjacks()
    {
        Console.WriteLine($"{Name} is eating flapjacks");
        while (flapjackStack.Count > 0)
        {
            Console.WriteLine(
                $"{Name} ate a {flapjackStack.Pop().ToString().ToLower()} flapjack");
        }
    }
}
```

Here's the stack of Flapjack enums. It gets filled up when the top-level statements call TakeFlapjack with random flapjacks, and drained when it calls the EatFlapjacks method.

The TakeFlapjack method just pushes a flapjack onto the stack.



there are no Dumb Questions

Q: What happens if I try to get an object from a Dictionary using a key that doesn't exist?

A: If you pass a Dictionary a key that doesn't exist, it will throw an exception. For example, if you add this code to a console app:

```
Dictionary<string, string> dict =
        new Dictionary<string, string>();

string s = dict["This key doesn't exist"];
```

you'll get the exception "System.Collections.Generic.KeyNotFoundException" with the message: *The given key 'This key doesn't exist' was not present in the Dictionary."*

Conveniently, the exception includes the key—or, more specifically, the string that the key's ToString method returns. That's really useful if you're trying to debug a problem in a program that accesses a Dictionary thousands of times.

Q: Is there a way to avoid that exception—like if I don't know if a key exists?

A: Yes, there are two ways to avoid a KeyNotFoundException. One way is to use the Dictionary.ContainsKey method. You pass it the key you want to use with the Dictionary, and it returns true only if the key exists. The other way is to use Dictionary.TryGetValue, which lets you do this:

```
if (dict.TryGetValue("Key", out string value))
{
    // do something
}
```

That code does exactly the same thing as this:

```
if (dict.ContainsKey("Key"))
{
    string value = dict["Key"];
    // do something
}
```

Bullet Points

- Lists, arrays, and other collections implement the **IEnumerable<T> interface**, which supports iteration over a nongeneric collection.
- A **`foreach` loop** works with any class that implements IEnumerable<T>, which includes a method to return an Enumerator object that lets the loop iterate through its contents in order.
- **Covariance** is C#'s way of letting you implicitly convert a subclass reference to its superclass.
- Covariance is useful when you need to pass a collection of objects to a method that only works with the class they inherit from. For example, covariance allows you to use **normal assignment to upcast** a List<Subclass> to an IEnumerable<Superclass>.
- A **Dictionary<TKey, TValue>** is a collection that stores a set of key/value pairs, and lets you use keys to look up their associated values.
- Dictionary keys and values can be **different types**. Every **key must be unique** in the Dictionary, but values can be duplicated.
- The Dictionary class has a **Keys property** that returns an iterable sequence of keys.
- A **Queue<T>** is a first-in, first-out collection with methods to enqueue an item at the end of the queue, and dequeue the item at the front of the queue.
- A **Stack<T>** is a last-in, first-out collection with methods to push an item onto the top of the stack and pop an item off of the top of the stack.
- The Stack<T> and Queue<T> classes **implement IEnumerable<T>**, and can easily be converted to Lists or other collection types.
- A **collection initializer** lets you initialze a List, Array, Dictionary, or any IEnumerable that has an Add method.

CollectionView is a MAUI control built for displaying collections

When you're building visual apps, it's very common to display a list of things. That's why .NET MAUI has a really convenient control called **CollectionView**, which displays the contents of a collection in a box, adding scrollbars to let your user scroll through the list if there are too many items to fit. Let's create an app to learn more.

Do this!

① **Create a new .NET MAUI app called MauiCards.**
For now, we'll display a familiar array of birds. But later on, we'll replace them with cards, and later we'll be reusing this code, so ***make sure you use the name MauiCards*** for your app.

② **Replace everything inside the <VerticalStackLayout> with a CollectionView.**
Open *MainPage.xaml* and delete everything between the opening and closing `VerticalStackLayout` tags. Replace it with this CollectionView inside a Border. CollectionView has a HeightRequest, which sets its height:

```
<Border>
    <CollectionView x:Name="MyItems" HeightRequest="125" />
</Border>
```

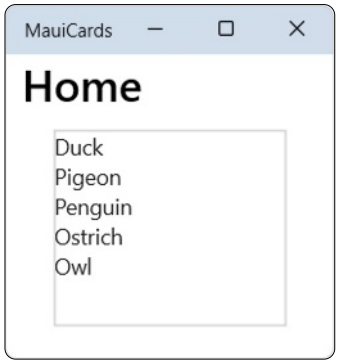
③ **Modify the MainPage constructor to add items to the CollectionView.**
Open *MainPage.xaml.cs* and delete everything inside of the MainPage class except its constructor.

Then flip back to the end of Chapter 2 and find the code where you added birds to a Picker control. You'll do exactly the same thing with this CollectionView instead. Add this code to the MainPage constructor:

```
public partial class MainPage : ContentPage
{
    private string[] birds = new string[] {
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    };

    public MainPage()
    {
        InitializeComponent();

        MyItems.ItemsSource = birds;
    }
}
```

Back in Chapter 2 you did something really similar to this to set the items in a Picker control. You can use a CollectionView in exactly the same way.

Make sure you delete any other code so your MainPage class matches ours.



④ **Run your app.**
Congratulations, you're displaying the contents of an array in a CollectionView! Technically, an array is not a .NET collection, but a CollectionView control can render the contents of an array just as easily as it can render the contents of a List.

ObservableCollection is a collection made for data binding

The ObservableCollection<T> class is a collection that's specially built for data binding, especially for displaying items in a CollectionView. It implements the familiar IEnumerable<T> and ICollection<T> interfaces (just like List<T>). It also implements the interfaces that let MAUI data binding know when its contents have changed.

1 **Add a button to your page to add a bird to the CollectionView.**

Modify *MainPage.xaml* to add a Button control just above the closing `</VerticalStackLayout>` tag:

```
<Button Text="Add an item" Clicked="Button_Clicked" />
```

2 **Create a new ObservableCollection and use it as the CollectionView items source.**

Modify *MainPage.xaml.cs* to add a Button control just above the closing `</VerticalStackLayout>` tag:

```
private ObservableCollection<string> myItems = new ObservableCollection<string>();

public MainPage()
{
    InitializeComponent();

    MyItems.ItemsSource = myItems;
}
```

ObservableCollection is in the System.Collections.ObjectModel namespace. Use the Quick Actions menu to *add a* `using` *directive* to the top of your *MainPage.xaml.cs* file:

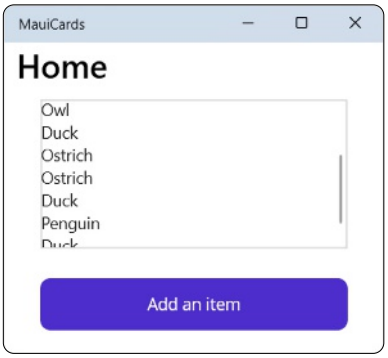
```
using System.Collections.ObjectModel;
```

3 **Make the button's event handler method add random birds to the collection.**

The ObservableCollection class has many of the same methods as the List class, including Add, Sort, and Clear. Modify the button's event handler method so it calls the Add method to add a random bird:

```
private void Button_Clicked(object sender, EventArgs e)
{
    myItems.Add(birds[Random.Shared.Next(0, birds.Length)]);
}
```

Now run your app and click the button a few times. Every time it adds a bird, the CollectionView updates.



You used the *HeightRequest property* to set the height of the CollectionView, so it adds a scroll bar if there are more items in the collection than fit in its height. Without this property, the CollectionView will automatically expand to fit the contents of the collection.

If you click the button many times, the birds scroll off the bottom and you need to scroll to see them. Add this property to the CollectionView so it automatically scrolls as items are added:

```
ItemsUpdatingScrollMode="KeepLastItemInView"
```

At the time we're writing this, there's a bug in MAUI on macOS where the CollectionView ignores the KeepLastItemInView option.

Add your Card class to the project

Let's make your CollectionView control work with objects, not just strings—specifically, card objects. You created a Card class, Suits and Values enums, and a CardComparerByValue class. Now you'll reuse them in a MAUI app.

1. **Add your Card class, CardComparerByValue class, Suits enum, and Values enum.**
The CollectionView control automatically calls the ToString method for any item that it displays, so make sure you use the version of the Card class ***that has the ToString method***. Add the existing files to your project just like you did in Chapter 3 (in Visual Studio: right-click on the project and choose Add >> Existing Item and add each file; in VSCode: right-click on the project and choose Reveal in Explorer/Finder and drag the files onto the project). Make sure the classes and enums are in the **MauiCards namespace**.

Open each file and make sure it has the correct namespace declaration.

2. **Replace the birds field with an ObservableCollection.**
Delete the birds field from your MainPage class. Then **replace the myItems declaration** with this:

```
private ObservableCollection<Card> myItems = new ObservableCollection<Card>();
```

3. **Modify the Button_Clicked event handler to add a random card.**
Here's the new code for the event handler:

```
private void Button_Clicked(object sender, EventArgs e)
{
    myItems.Add(
       new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4)));
}
```

Now run your app. When you click the button, it adds a random card to the CollectionView.

4. **Use the SelectionMode property to let your user select a card.**
The CollectionView control has a **SelectionMode property** that lets your user select either a single item or multiple items in the collection. Modify your XAML to add a Label control to display the selected card, and add the `SelectionMode="Single"` property so the user can select a card. When the user selects an item in a CollectionView, it calls the SelectionChanged event, so **add a SelectionChanged event handler too**.

```
<Label x:Name="SelectedCard" Text="No card selected"/>
<Border>
    <CollectionView x:Name="MyItems" HeightRequest="125" SelectionMode="Single"
                    SelectionChanged="MyItems_SelectionChanged"
                    ItemsUpdatingScrollMode="KeepLastItemInView" />
</Border>
```

The SelectionChanged event handler method uses the SelectedItem property to access the selected item:

```
private void MyItems_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (MyItems.SelectedItem == null)
        SelectedCard.Text = $"No card selected";
    else
        SelectedCard.Text = $"You selected {MyItems.SelectedItem}";
}
```

Run your app and select cards, or Ctrl-Click to deselect a card. It will show the selected card in the Label.

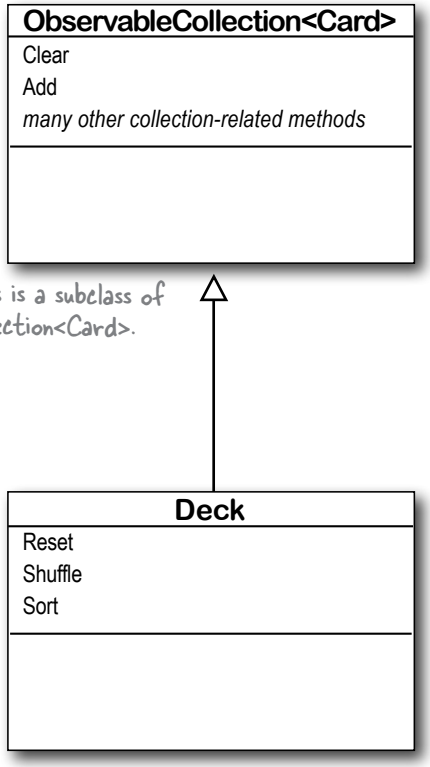
Exercise

Create a Deck class that extends ObservableCollection.

You learned all about inheritance in Chapter 6. Now it's time to apply that knowledge to create a class that represents a deck of cards. **Add a Deck class** to your project. Make sure that it includes this `using` directive:

```
using System.Collections.ObjectModel;
```

Make the Deck class **extend ObservableCollection<Card>** so it inherits all of the collection-related methods, including the Clear and Add methods.




```
class Deck : ObservableCollection<Card>
{
  /// <summary>
  /// The constructor resets the 52-card deck
  /// </summary>
  public Deck() {
     Reset();
  }

  /// <summary>
  /// Clears the deck, then loops through suits and
  /// values, and adds each card to the 52-card deck
  /// </summary>
  public void Reset() { ... }

  /// <summary>
  /// Creates a copy of the deck, clears the deck, and
  /// uses a while loop to move a random card from
  /// the copy to the deck and remove it from the copy
  /// </summary>
  public void Shuffle() { ... }

  /// <summary>
  /// Creates a copy of the deck, sorts it using
  /// CardComparerByValue, then moves each card from
  /// the copy back to the deck
  /// </summary>
  public void Sort() { ... }
}
```

The Deck class is a subclass of ObservableCollection<Card>.

You can create a copy of the deck by using this List<T> constructor:

```
List<Card> copy = new
  List<Card>(this);
```

Then call Deck.Clear(), and use a loop to move cards from the copy back to the Deck.

Modify your *MainPage.xaml* **and** *MainPage.xaml.cs*

Replace the myItems declaration so it creates a Deck object instead of an array of strings with birds:

```
private Deck myItems = new Deck();
```

This replaces the myItems field in MainPage.xaml.cs.

Modify your XAML to add buttons to call the Deck methods:

```
<Button Text="Shuffle the deck" Clicked="Shuffle_Clicked"/>

<Button Text="Sort the deck" Clicked="Sort_Clicked"/>

<Button Text="Reset the deck" Clicked="Reset_Clicked"/>

<Button Text="Clear the deck" Clicked="Clear_Clicked"/>
```

Make the event handlers for these buttons call the method in the Deck.

You'll add these new buttons to your page to call the Shuffle, Sort, Reset, and Clear methods on the Deck object.

Exercise Solution

Here's the Deck class, including the `using` directive to use the namespace ObservableCollection<T> is in:

```
using System.Collections.ObjectModel;

class Deck : ObservableCollection<Card>
{
    public Deck()
    {
        Reset();
    }

    public void Reset()
    {
        Clear();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                Add(new Card((Values)value, (Suits)suit));
    }

    public void Shuffle()
    {
        List<Card> copy = new List<Card>(this);
        Clear();
        while (copy.Count > 0)
        {
            int index = Random.Shared.Next(copy.Count);
            Card card = copy[index];
            copy.RemoveAt(index);
            Add(card);
        }
    }

    public void Sort()
    {
        List<Card> sortedCards = new List<Card>(this);
        sortedCards.Sort(new CardComparerByValue());
        Clear();
        foreach (Card card in sortedCards)
        {
            Add(card);
        }
    }
}
```

Does your CollectionView show "MauiCard.Card" instead of a card name? Make sure your Card class has a ToString method that returns Name.

This nested for loop goes through each of the suits, and for each suit it loops through all of the cards and adds them to the Deck.

This while loop picks a random card from the copy, adds it to the Deck, and then removes it from the copy, repeating until the copy is empty.

ObservableCollection doesn't have a Sort method—that's why you had to copy the cards to a List, sort it, and copy the cards back to the deck.

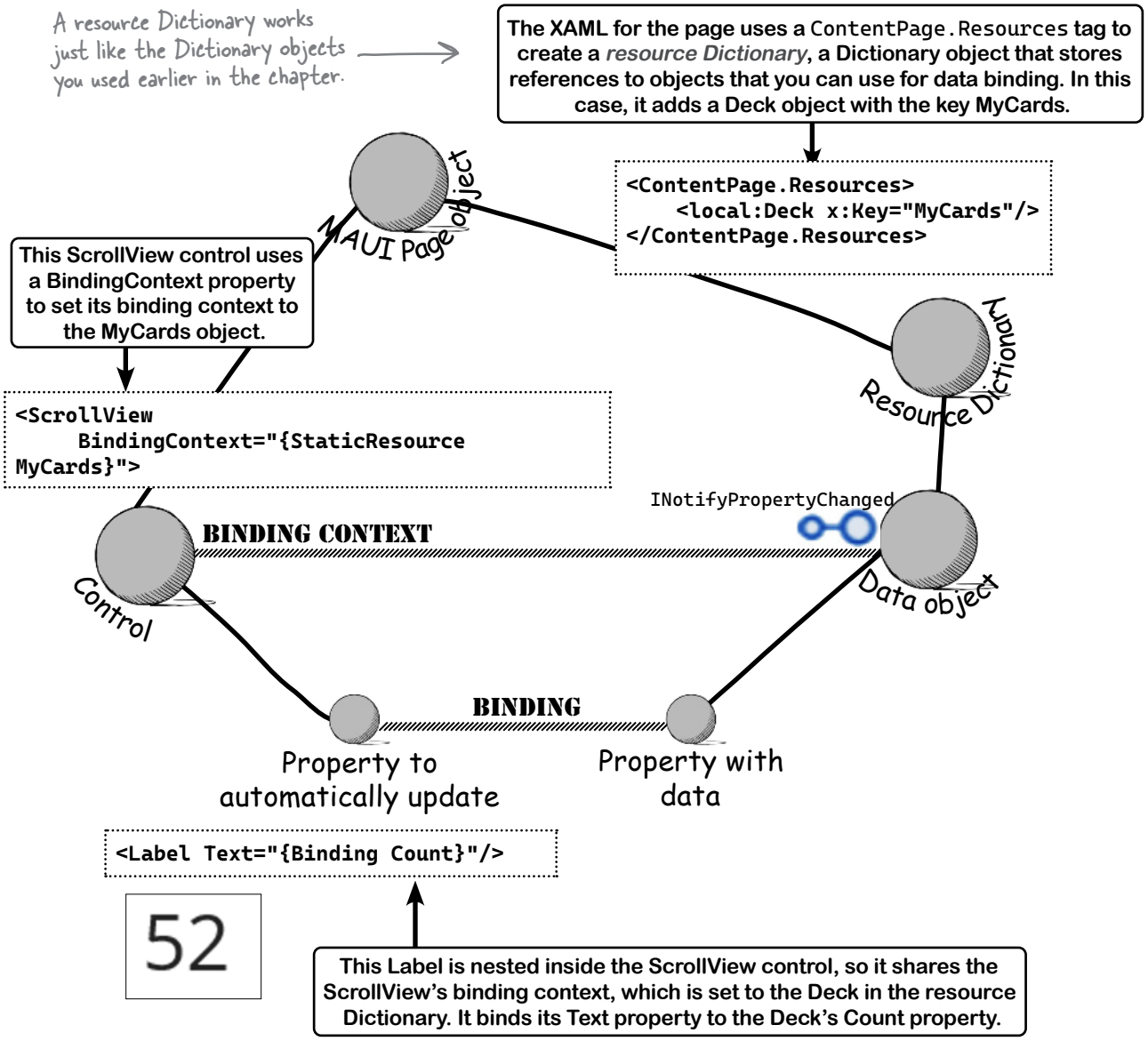
Here are the event handler methods for the four buttons you added to the XAML:

```
private void Shuffle_Clicked(object sender, EventArgs e) { myItems.Shuffle(); }

private void Sort_Clicked(object sender, EventArgs e) { myItems.Sort(); }

private void Reset_Clicked(object sender, EventArgs e) { myItems.Reset(); }

private void Clear_Clicked(object sender, EventArgs e) { myItems.Clear(); }
```

Use XAML to instantiate your objects for data binding

Your app uses C# code to create an instance of your Deck class and sets the CollectionView control's ItemsSource to set up data binding. In previous projects, you set the binding context for your data binding by writing C# code to set the page's BindingContext property.

There's another way to do **data binding in MAUI directly in XAML** without setting properties in C# code. Doing binding directly in the XAML code can be really useful when you're binding a lot of objects, so you don't have to jump back and forth between the XAML and C# code. Here's how it works:



Modify your app to use a resource Dictionary

Let's get some practice using a resource Dictionary by removing the Deck field from the C# code in your *MainPage.xaml.cs* file and instantiating it in your XAML instead.

Do this!

1 **Remove the Deck field from your code-behind.**

Start by removing the myItems field and modifying the constructor to remove the line that sets the CollectionView control's ItemsSource property.

```
public partial class MainPage : ContentPage
{
    private Deck myItems = new Deck();

    public MainPage()
    {
        InitializeComponent();

        MyItems.ItemsSource = myItems;
    }
```

This is why we asked you to name your app MauiCards. If your app has a different namespace, modify the xmlns:local line to use that namespace instead.

2 **Add the resource Dictionary to your XAML.**

Add this **`xmlsns:local` property** to your `ContentPage` tag so it knows about the MauiCards namespace:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MauiCards"
             x:Class="MauiCards.MainPage">
```

Then add this `ContentPage.Resources` tag right underneath the `ContentPage` tag above the opening `ScrollView` tag:

```
    <ContentPage.Resources>
        <local:Deck x:Key="MyCards"/>
    </ContentPage.Resources>
```

Now your app has a resource Dictionary that contains a Deck object with the key `MyCards`.

Declare a namespace in your <ContentPage> tag

Take a closer look at the XAML you just added. You added an xmlns:local property to the ContentPage tag that had the value **`"clr-namespace:MauiCards"`**—this defined "local" in your XAML to mean the MauiCards namespace. When you added a **`local:Deck`** resource tag, you told MAUI to instantiate a class called Deck that's in the "local" namespace, and your Deck class is in the MauiCards namespace. When you use a class from the same namespace as your app, you'll typically name it "local"—but you can put any name after "xmlns:" to look for classes in other namespaces.

③ Add a Label that binds to the Deck's Count property.

Since Deck extends the ObservableCollection class, it inherits the Count property—and you can use that with data binding, just like you did in Chapter 7.

In Chapter 7, you modified the code-behind in the *MainPage.xaml.cs* file to set the BindingContext property in the MainPage constructor. Now that you have a resource Dictionary, you can use the **StaticResource markup** to set the binding context directly in XAML.

You're going to add a Label that displays this text: **52 cards in the deck**—except instead of the number 52, you'll use data binding. Luckily, data binding lets us use **composite formatting**. You'll bind the Text field to `"{Binding Count, StringFormat='There are {0} cards in the deck'}"`—this causes the binding to replace `{0}` with the specified value in the binding context—in this case, `Count`.

Here's the XAML to add just below the opening `<VerticalStackLayout>` tag:

```
<Label BindingContext="{StaticResource MyCards}"
       Text="{Binding Count, StringFormat='There are {0} cards in the deck'}" />
```

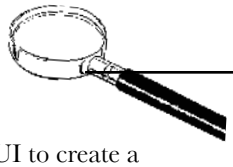
The label will be set to this string, replacing {0} with the value of Count.

④ Set the CollectionView control's ItemsSource directly in the XAML.

Find the XAML for your CollectionView control and add this ItemsSource property. Use the StaticResource markup to tell it that the source for the items in the list is your Deck resource.

```
<CollectionView x:Name="MyItems" HeightRequest="125" SelectionMode="Single"
                SelectionChanged="MyItems_SelectionChanged"
                ItemsUpdatingScrollMode="KeepLastItemInView"
                ItemsSource="{StaticResource MyCards}" />
```

Using a Resource Dictionary Up Close



When you added the `<ContentPage.Resources>` tag to your ContentPage, you told MAUI to create a **resource Dictionary** for the page. That's a Dictionary to hold objects that the XAML elements on your page can access, and it works just like the Dictionary class you've been using—including the **TryGetValue method**.

You'll often need your code-behind to work with objects in the resource Dictionary. For example, your app has buttons that need to call methods on the Deck resource stored in the Dictionary. You can access the resource Dictionary using the ContentPage's Resources property. You'll use this code to fetch the Deck from the Dictionary:

```
if (Resources.TryGetValue("MyCards", out object myCards) && myCards is Deck deck)
    // now the Deck variable contains a reference to the MyDeck resource
```

Take a close look at what's happening in this line of code. It uses the Resources property to access the resource Dictionary, then calls its TryGetValue method to get the Deck reference. If it finds the Deck reference, it uses is to cast it to a Deck. Take a little time to understand what's going on here.

Modify the event handlers to use the resource Dictionary

We just showed you a line of code that gets a reference to the Deck object in the page's resource Dictionary. Let's use that code to get your app's button event handlers to work with the Deck object in the page's resource Dictionary.

- Remove the myItems field from your MainPage class and **add a method called GetDeckFromResources** that uses Resources.TryGetValue to get the reference to the Deck object from the Resources Dictionary.
- Modify the five Button event handlers to **replace myItems with GetDeckFromResources**().

Once you've done that, the code-behind for your *MainPage.xaml.cs* file should look like this:

```
namespace MauiCards;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private Deck GetDeckFromResources()
    {
        if (Resources.TryGetValue("MyCards", out object myCards) && myCards is Deck deck)
            return deck;
        else
            return new Deck();
    }

    private void Button_Clicked(object sender, EventArgs e)
    {
        GetDeckFromResources().Add(
            new Card((Values)Random.Shared.Next(1, 14), (Suits)Random.Shared.Next(4)));
    }

    private void Shuffle_Clicked(object sender, EventArgs e) {
        GetDeckFromResources().Shuffle();
    }

    private void Sort_Clicked(object sender, EventArgs e) {
        GetDeckFromResources().Sort();
    }

    private void Reset_Clicked(object sender, EventArgs e) {
        GetDeckFromResources().Reset();
    }

    private void Clear_Clicked(object sender, EventArgs e) {
        GetDeckFromResources().Clear();
    }
    // The MyItems_SelectionChanged method has not changed
}
```

The GetDeckFromResources method uses Resources.TryGetValue to get the Deck object from the page's resource Dictionary, then uses "is" to safely cast it to a Deck reference.

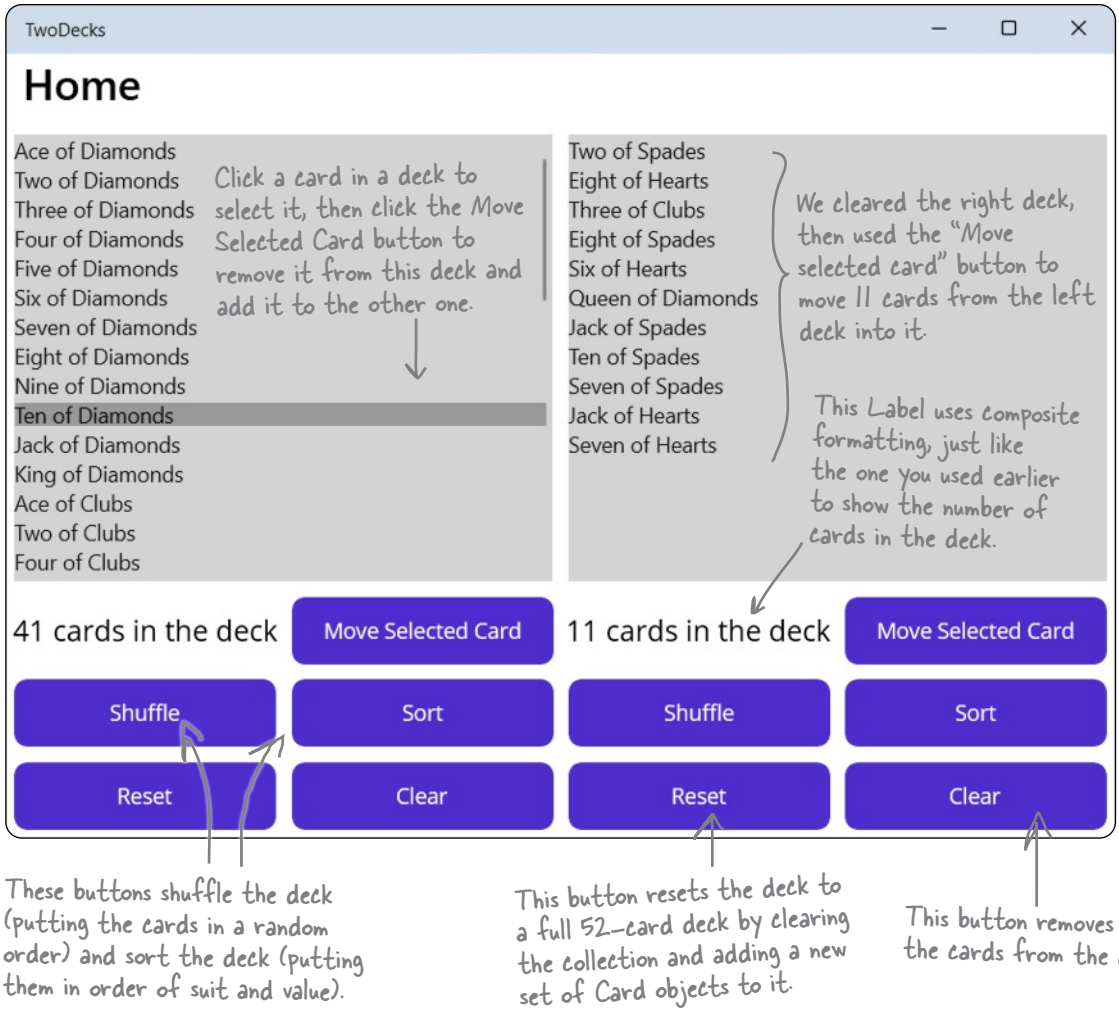
Resources.TryGetValue returns an Object reference, so we need to use is to safely cast it to a Deck reference. This method should never reach this statement—but we included it so the app doesn't throw an exception just in case it does. Did we make the right choice?

The myItems field that you removed from the MainPage class has a reference to a Deck object. The GetDeckFromResources method returns a reference to the Deck in the page's resource Dictionary, so replacing myItems with a call to that method makes the page work just like it did before.

Use what you've learned to build an app with two decks

You've just used a CollectionView control, an ObservableCollection (which you extended to create your Deck class), created a resource Dictionary with a Deck object, done all of the data binding directly in your XAML code, and written C# code to access that object in the resource Dictionary.

This next exercise is about ***really getting those new C# concepts and tools into your brain***. You'll create an app that has two decks of cards, with buttons that let you shuffle, sort, reset, and clear the decks. You'll also take advantage of a feature in the CollectionView control that lets you select an item in the list to create buttons to move cards between the decks.



In this next exercise, you'll take the ideas and tools you just used and apply them to a new project. This is a great way to get them to stick in your brain.

Exercise

You've just learned about CollectionView controls and more about data binding. Combine what you've learned with what you already know about building .NET MAUI apps to build an app that manages two decks.

Create a .NET MAUI project called TwoDecks and add Deck and the other card-related classes and enums

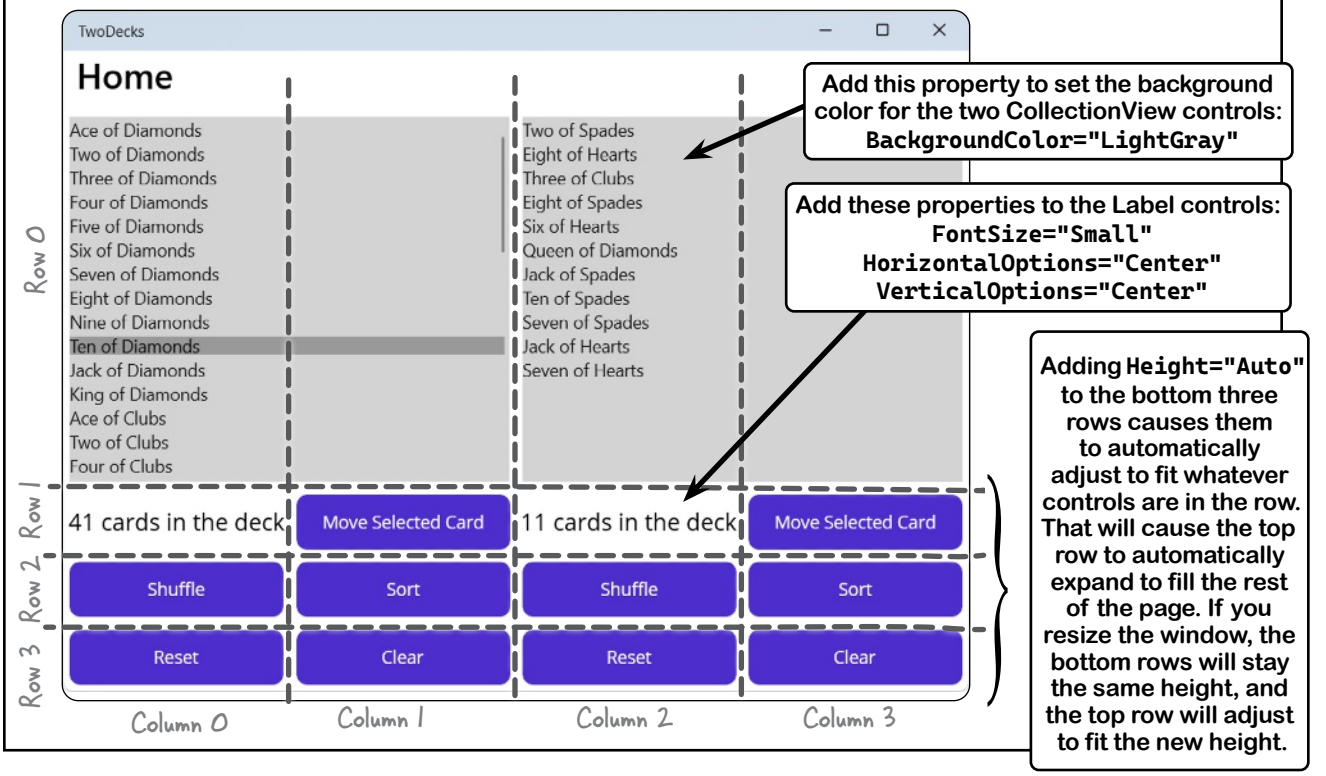
The first thing you'll do is add the Suits and Values enums and the Card, Deck, and CardComparerByValue classes. But ***do not change the namespaces***—keep them in the MauiCards namespace. Start by **creating a new folder** in your project: right-click on the project in the Solution Explorer and choose Add >> New Folder (Visual Studio) or New Folder (VSCode). Name your new folder **MauiCards**.

In Visual Studio, right-click on the new MauiCards folder and choose the Add >> Existing Item (Shift+Alt+A) to add each file to the folder. In VSCode, right-click on the MauiCards folder and choose "Reveal in Explorer/Finder" and copy the files into the folder. Copy the Suits and Values enums and the Deck, Card, and CardComparerByValue classes. **Do not** change the namespace—we'll reuse the MauiCards namespace.

Rebuild your project and make sure that it does not have any warnings.

Use a grid with four columns and four rows to lay out the page

Delete everything inside the `<ContentPage>` tag and replace it with a `<Grid>` with four columns and four rows. The four columns are equal widths, so none of the `<ColumnDefinition/>` tags should have a Width property. Don't add a Height property to the top `<RowDefinition/>` tag, but give the bottom three a Height="Auto" property.



Exercise

Add a resource Dictionary to your page with two Deck objects

In the last project, you added a resource Dictionary to your page with a Deck object. You'll add a resource Dictionary to this page too, but this time it will create two Deck objects: one reference will have the key LeftDeck, and the other reference will have the key RightDeck.

Since you copied *Deck.cs* and the other card-related files from the MauiCards project, they should all have the `namespace MauiCards` declaration at the top. **Add a xmlns:mauiCards property** to your `ContentPage` tag, which will let you use the **`mauiCards:`** prefix to create objects in the MauiCards namespace when you declare your resource Dictionary. Replace the XAML in *MainPage.xaml.cs* with this, then add controls nested in the Grid:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:mauiCards="clr-namespace:MauiCards"
             x:Class="TwoDecks.MainPage">

    <ContentPage.Resources>
        <mauiCards:Deck x:Key="LeftDeck"/>
        <mauiCards:Deck x:Key="RightDeck"/>
    </ContentPage.Resources>

    <Grid>

       The contents of the page will go here

    </Grid>

</ContentPage>
```

Your Deck class is in the MauiCards namespace, so you'll add this mauiCards prefix to use it.

Here's where you'll declare your resource Dictionary with two Deck references. They have the keys LeftDeck and RightDeck.

This is where the controls nested in the Grid will go. You'll also need to add the row and column definitions here.

Modify your code-behind to get Deck references from the resource Dictionary so the buttons can use them

You'll be using the Deck type in your code, so make sure you add this using directive at the top of MainPage.xaml.cs:

```
using MauiCards;
```

Add this method to your code-behind. You had a really similar method in your last project, and this one is almost identical—the only difference is that it takes the key as a parameter so you can get either Deck from the Dictionary.

```
private Deck GetDeckFromResources(String key)
{
    if (Resources.TryGetValue(key, out object resource) && resource is Deck deck)
        return deck;
    else
        return new Deck();
}
```

There's one more piece of the puzzle: you need a way to figure out what card is selected so you can make the buttons that move a card from one deck to the other work. Use x:Name to **name your CollectionViews** LeftCards and RightCards. If a card is selected in the left CollectionView, **`LeftCards.SelectedItem is Card card`** will return true. If it does, call `GetDeckFromResources("RightDeck").Add(card)` to add the card to the right deck, then call `GetDeckFromResources("LeftDeck").Remove(card)` to remove it from the left deck. Then do the reverse for the button to move a card from the right deck to the left, adding it to the left deck and removing it from the right.

Exercise Solution

Here's the XAML for the page in *MainPage.xaml.cs*.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:mauiCards="clr-namespace:MauiCards"
             x:Class="TwoDecks.MainPage">

    <ContentPage.Resources>
        <mauiCards:Deck x:Key="LeftDeck"/>
        <mauiCards:Deck x:Key="RightDeck"/>
    </ContentPage.Resources>

    <Grid>

        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <CollectionView x:Name="LeftCards" ItemsSource="{StaticResource LeftDeck}"
               Grid.ColumnSpan="2" Margin="5" SelectionMode="Single"
               BackgroundColor="LightGray" />

        <CollectionView x:Name="RightCards" ItemsSource="{StaticResource RightDeck}"
               Grid.Column="2" Grid.ColumnSpan="2" Margin="5" SelectionMode="Single"
               BackgroundColor="LightGray"/>

        <Label BindingContext="{StaticResource LeftDeck}" Grid.Row="1"
               Text="{Binding Count, StringFormat='{0} cards in the deck'}"
               FontSize="Small" HorizontalOptions="Center" VerticalOptions="Center" />
```

This xmlns property in the ContentPage tag sets up the "mauiCards" namespace, which contains Deck and the other card-related classes.

Your page has a resource Dictionary with two Deck objects, with the keys LeftDeck and RightDeck.

The page has four equal-sized columns.

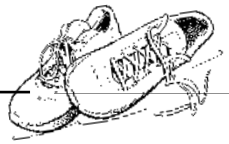
The CollectionView is in the top row, and the buttons are in the bottom four rows. Setting the button row heights to "Auto" causes them to shrink so they're exactly the size of the buttons, leaving the top row to expand to fill up the rest of the grid.

The two CollectionView controls span two columns, so they're twice as wide as each button.

These are just like the ones you used in the Two Decks project.

This label uses the BindingContext property to bind to the left Deck object and binds its Text property using a StringFormat with composite formatting.

Now that you're building more complex apps, there's a good chance that your code looks a little different than ours. That's OK! If your app works, you did it right. It's worth spending a few minutes looking at our solution too—seeing the problem solved in more than one way is a great way to get these concepts into your brain.



Exercise Solution

```
        <Button Text="Move Selected Card" Grid.Row="1" Grid.Column="1"
                SemanticProperties.Hint="Moves the selected card from left to right"
                Margin="5" Clicked="MoveLeftDeck_Clicked" />

        <Button Text="Shuffle" Grid.Row="2"
                SemanticProperties.Hint="Shuffles the left deck"
                Margin="5" Clicked="ShuffleLeftDeck_Clicked" />

        <Button Text="Sort" Grid.Row="2" Grid.Column="1"
                SemanticProperties.Hint="Sorts the left deck"
                Margin="5" Clicked="SortLeftDeck_Clicked" />

        <Button Text="Reset" Grid.Row="3"
                SemanticProperties.Hint="Resets the left deck"
                Margin="5" Clicked="ResetLeftDeck_Clicked" />

        <Button Text="Clear" Grid.Row="3" Grid.Column="1"
                SemanticProperties.Hint="Clears the left deck"
                Margin="5" Clicked="ClearLeftDeck_Clicked" />

        <Label BindingContext="{StaticResource RightDeck}" Grid.Row="1" Grid.Column="2"
              FontSize="Small" HorizontalOptions="Center" VerticalOptions="Center"
               Text="{Binding Count, StringFormat='{0} cards in the deck'}" />

        <Button Text="Move Selected Card" Grid.Row="1" Grid.Column="3"
                SemanticProperties.Hint="Moves the selected card from right to left"
                Margin="5" Clicked="MoveRightDeck_Clicked" />

        <Button Text="Shuffle" Grid.Row="2" Grid.Column="2"
                SemanticProperties.Hint="Shuffles the right deck"
                Margin="5" Clicked="ShuffleRightDeck_Clicked" />

        <Button Text="Sort" Grid.Row="2" Grid.Column="3" Margin="5"
                SemanticProperties.Hint="Sorts the right deck"
                Clicked="SortRightDeck_Clicked" />

        <Button Text="Reset" Grid.Row="3" Grid.Column="2" Margin="5"
                SemanticProperties.Hint="Resets the right deck"
                Clicked="ResetRightDeck_Clicked" />

        <Button Text="Clear" Grid.Row="3" Grid.Column="3" Margin="5"
                SemanticProperties.Hint="Clears the right deck"
                Clicked="ClearRightDeck_Clicked" />

    </Grid>

</ContentPage>
```

These are the bottom four buttons on the left side of the page. They're in the left two columns.

Both labels use bindings with composite formatting.

Did you remember to include semantic properties? Accessibility is really important, and you should get in the habit of making your apps accessible.

The labels and buttons on the right side of the page are almost the same as the ones on the left.

Exercise Solution

Here's the C# from the code-behind for the page. Make sure to include the class declaration, constructor, and *using directive*: `using MauiCards;`

```
private Deck GetDeckFromResources(String key)
{
    if (Resources.TryGetValue(key, out object resource) && resource is Deck deck)
        return deck;
    else
        return new Deck();
}

private void ShuffleLeftDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("LeftDeck").Shuffle();
}

private void SortLeftDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("LeftDeck").Sort();
}

private void ClearLeftDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("LeftDeck").Clear();
}

private void ResetLeftDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("LeftDeck").Reset();
}

private void MoveLeftDeck_Clicked(object sender, EventArgs e)
{
    if (LeftCards.SelectedItem is Card card)
    {
        GetDeckFromResources("RightDeck").Add(card);
        GetDeckFromResources("LeftDeck").Remove(card);
    }
}

private void ShuffleRightDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("RightDeck").Shuffle();
}

private void SortRightDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("RightDeck").Sort();
}

private void ClearRightDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("RightDeck").Clear();
}

private void ResetRightDeck_Clicked(object sender, EventArgs e)
{
    GetDeckFromResources("RightDeck").Reset();
}

private void MoveRightDeck_Clicked(object sender, EventArgs e)
{
    if (RightCards.SelectedItem is Card card)
    {
        GetDeckFromResources("LeftDeck").Add(card);
        GetDeckFromResources("RightDeck").Remove(card);
    }
}
```

We gave you this method to get a Deck from the resources.

These Clicked event handlers work just like the ones in your previous app.

Here's the Clicked event handler for the button that moves the selected card from the left deck to the right one. It works exactly the way we described in the instructions.

The event handlers for the buttons on the right work just like the ones on the left.

Unity Lab #4

User Interfaces

In the last Unity Lab you started to build a game, using a prefab to create GameObject instances that appear at random points in your game's 3D space and fly in circles. This Unity Lab picks up where the last one left off, allowing you to apply what you've learned about interfaces in C# and more.

Your program so far is an interesting visual simulation. The goal of this Unity Lab is to **finish building the game**. It starts off with a score of zero. Billiard balls will start to appear and fly around the screen. When the player clicks on a ball, the score goes up by 1 and the ball disappears. More and more balls appear; once 15 balls are flying around the screen, the game ends. For your game to work, your players need a way to start it and to play again once the game is over, and they'll want to see their score as they click on the balls. So you'll add a **user interface** that displays the score in the corner of the screen, and shows a button to start a new game.

Add a score that goes up when the player clicks a ball

In Unity Lab 3, you built an interesting simulation. Now it's time to turn it into a game. **Add a new field** to the GameController class to keep track of the score—you can add it just below the OneBallPrefab field:

```
public int Score = 0;
```

Next, **add a method called ClickedOnBall to the GameController class**. This method will get called every time the player clicks on a ball:

```
public void ClickedOnBall()
{
    Score++;
}
```

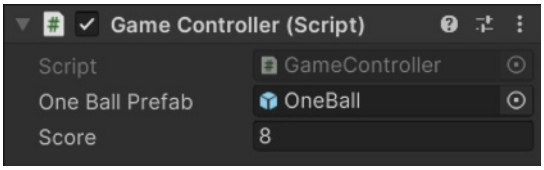
Unity makes it really easy for your GameObjects to respond to mouse clicks and other input. If you add a method called OnMouseDown to a script, Unity will call that method any time the GameObject it's attached to is clicked. **Add this method to the OneBallBehaviour class**:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    controller.ClickedOnBall();
    Destroy(gameObject);
}
```

We learned about generics in Chapter 8. Here's another example: passing the type to the generic GetComponent method causes it to return a GameController.

The first line of the OnMouseDown method gets the instance of the GameController class, and the second line calls its ClickedOnBall method, which increments its Score field.

Now run your game. Click on Main Camera in the hierarchy and watch its Game Controller (Script) component in the Inspector. Click on some of the rotating balls—they'll disappear and the Score will go up.



If you don't see the Score go up when you click each ball, make sure you added the OnMouseDown method to OneBallBehaviour, not GameController.

there are no Dumb Questions

Q: Why do we use Instantiate instead of the `new` keyword?

A: Instantiate and Destroy are ***special methods that are unique to Unity***—you won't see them in your other C# projects. The Instantiate method isn't quite the same thing as the C# `new` keyword, because it's creating a new instance of a prefab, not a class. Unity does create new instances of objects, but it needs to do a lot of other things, like making sure that it's included in the `update` loop. When a GameObject's script calls Destroy(gameObject) it's telling Unity to destroy itself. The Destroy method tells Unity to destroy a GameObject—but not until after the `update` loop is complete.

Q: I'm not clear on how the first line of the OnMouseDown method works. What's going on there?

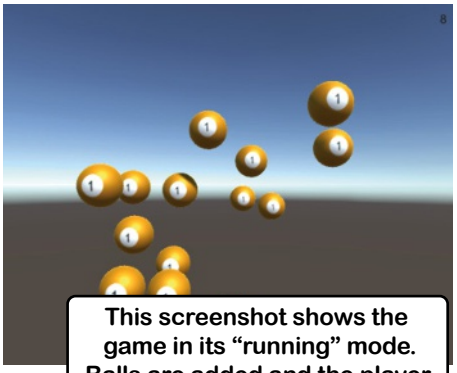
A: Let's break down that statement piece by piece. The first part should be pretty familiar: it declares a variable called `controller` of type GameController, the class that you defined in the script that you attached to the Main Camera. In the second half, we want to call a method on the GameController attached to the Main Camera. So we use Camera.main to get the Main Camera, and GetComponent<GameController>() to get the instance of GameController that we attached to it.

Add two different modes to your game

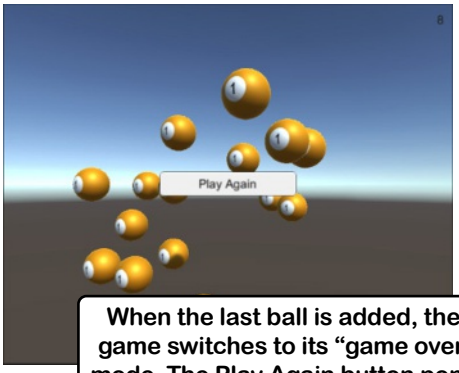
Start up your favorite game. Do you immediately get dropped into the action? Probably not—you're probably looking at a start menu. Some games let you pause the action to look at a map. Many games let you switch between moving the player and working with an inventory, or show an animation while the player is dying that can't be interrupted. These are all examples of **game modes**.

Let's add two different modes to your billiard ball game:

- **Mode #1: The game is running.** Balls are being added to the scene, and clicking on them makes them disappear and the score go up.
- **Mode #2: The game is over.** Balls are no longer getting added to the scene, clicking on them doesn't do anything, and a "Game over" banner is displayed.



This screenshot shows the game in its "running" mode. Balls are added and the player can click on them to score.



When the last ball is added, the game switches to its "game over" mode. The Play Again button pops up and no more balls get added.

You'll add two modes to your game. You already have the "running" mode, so now you just need to add a "game over" mode.

Here's how you'll add the two game modes to your game:

1. **Make GameController.AddABall pay attention to the game mode.**

   Your new and improved AddABall method will check if the game is over, and will only instantiate a new OneBall prefab if the game is not over.

2. **Make OneBallBehaviour.OnMouseDown only work when the game is running.**

   When the game is over, we want the game to stop responding to mouse clicks. The player should just see the balls that were already added continue to circle until the game restarts.

3. **Make GameController.AddABall end the game when there are too many balls.**

   AddABall also increments its NumberOfBalls counter, so it goes up by 1 every time a ball is added. If the value reaches MaximumBalls, it sets GameOver to true to end the game.

In this lab, you're building this game in parts, and making changes along the way. You can download the code for each part from the book's GitHub repository: *https://github.com/head-first-csharp/fifth-edition*

Add game mode to your game

Modify your GameController and OneBallBehaviour classes to **add modes to your game** by using a Boolean field to keep track of whether or not the game is over.

1 **Make GameController.AddABall pay attention to the game mode.**

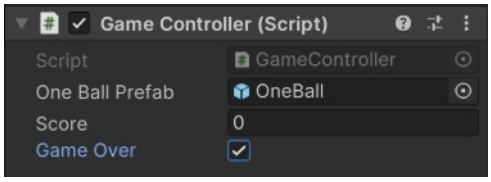
We want the GameController to know what mode the game is in. When we need to keep track of what an object knows, we use fields. Since there are two modes—running and game over—we can use a Boolean field to keep track of the mode. **Add the GameOver field** to your GameController class:

```
public bool GameOver = false;
```

The game should only add new balls to the scene if the game is running. Modify the AddABall method to add an `if` statement that only calls Instantiate if GameOver is not true:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
    }
}
```

Now you can test it out. Start your game, then **click on Main Camera** in the Hierarchy window.



Check the Game Over box while the game is running to toggle the GameController's GameOver field. If you check it while the game is running, Unity will reset it when you stop the game.

Set the GameOver field by unchecking the box in the Script component. The game should stop adding balls until you check the box again.

2 **Make OneBallBehaviour.OnMouseDown only work when the game is running.**

Your OnMouseDown method already calls the GameController's ClickedOnBall method. Now **modify OnMouseDown in OneBallBehaviour** to use the GameController's GameOver field as well:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Run your game again and test that balls disappear and the score goes up only when the game is not over.

3 **Make GameController.AddABall end the game when there are too many balls.**

The game needs to keep track of the number of balls in the scene. We'll do this by **adding two fields** to the GameController class to keep track of the current number of balls and the maximum number of balls:

```
public int NumberOfBalls = 0;
public int MaximumBalls = 15;
```

Every time the player clicks on a ball, the ball's OneBallBehaviour script calls GameController.ClickedOnBall to increment (add 1 to) the score. Let's also decrement (subtract 1 from) NumberOfBalls:

```
public void ClickedOnBall()
{
    Score++;
    NumberOfBalls--;
}
```

Now **modify the AddABall** method so that it only adds balls if the game is running, and ends the game if there are too many balls in the scene:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
        NumberOfBalls++;
        if (NumberOfBalls >= MaximumBalls)
        {
            GameOver = true;
        }
    }
}
```

The GameOver field is true if the game is over and false if the game is running. The NumberOfBalls field keeps track of the number of balls currently in the scene. Once it hits the MaximumBalls value, the GameController will set GameOver to true.

Now test your game one more time by running it and then clicking on Main Camera in the Hierarchy window. The game should run normally, but as soon as the NumberOfBalls field is equal to the MaximumBalls field, the AddABall method sets its GameOver field to true and ends the game.

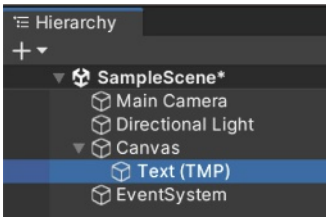
Once that happens, clicking on the balls doesn't do anything because OneBallBehaviour.OnMouseDown checks the GameOver field and only increments the score and destroys the ball if GameOver is false.

Your game needs to keep track of its game mode. Fields are a great way to do that.

Add a UI to your game

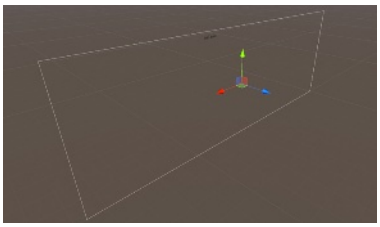
Almost any game you can think of—from Pac-Man to Super Mario Brothers to Grand Theft Auto 5 to Fortnite—features a **user interface (or UI)**. Some games, like Pac-Man, have a very simple UI that just shows the score, high score, lives left, and current level. Many games feature an intricate UI incorporated into the game's mechanics (like a weapon wheel that lets the player quickly switch between weapons). Let's add a UI to your game.

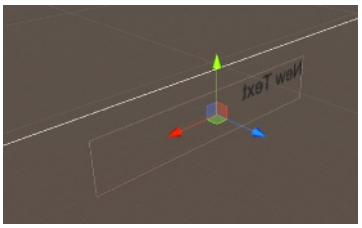
Choose UI >> Text - TextMeshPro from the GameObject menu to add a 2D Text GameObject to your game's UI. This adds a Canvas to the Hierarchy, and a Text under that Canvas:



The first time you add text to your scene, Unity will prompt you to *import TMP essentials*—make sure you do that. When you added the text to your scene, Unity automatically added Canvas and Text GameObjects. Click the triangle (▼) next to Canvas to expand or collapse it—the Text (TMP) GameObject will appear and disappear because it's *nested* under Canvas.

Double-click on Canvas in the Hierarchy window to focus on it. It's a 2D rectangle. Click on its Move Gizmo and drag it around the scene. It won't move! The Canvas that was just added will always be displayed, scaled to the size of the screen, and in front of everything else in the game.





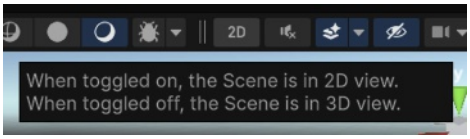
Did you notice an *EventSystem* in the Hierarchy? Unity automatically added it when you created the UI. It manages mouse, keyboard, and other inputs and sends them back to GameObjects—and it does all of that automatically, so you won't need to work directly with it.

Then double-click on Text to focus on it—the editor will zoom in, but the default text ("New Text") will be backward because the Main Camera is pointing at the back of the Canvas.

A *Canvas* is a two-dimensional GameObject that lets you lay out your game's UI. Your game's Canvas will have two GameObjects nested under it: the *Text* GameObject that you just added will be in the upper-right corner to display the score, and there's a *Button* GameObject to let the player start a new game.

Use the 2D view to work with the Canvas

The **2D button** at the top of the Scene window toggles 2D view on and off:



Click the 2D view—the editor flips around its view to show the Canvas head-on. **Double-click on Text (TMP)** in the Hierarchy window to zoom in on it.

Use the mouse wheel to zoom in and out in 2D view.

You can **click the 2D button to switch between 2D and 3D**. Click it again to return to the 3D view.

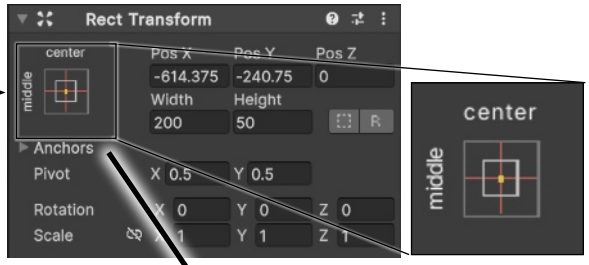
Set up the Text that will display the score in the UI

Your game's UI will feature one Text GameObject and one Button. Each of those GameObjects will be **anchored** to a different part of the UI. For example, the Text GameObject that displays the score will show up in the upper-right corner of the screen (no matter how big or small the screen is).

The Text is anchored to a specific point in the 2D Canvas.

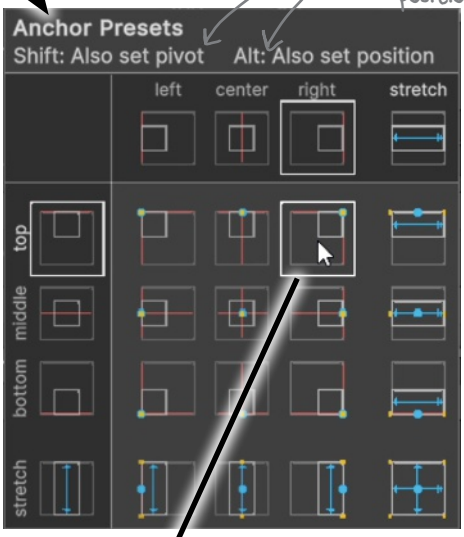
Click on Text in the Hierarchy window to select it, then look at the Rect Transform component. We want the Text in the upper-right corner, so **click the Anchors box** in the Rect Transform panel.

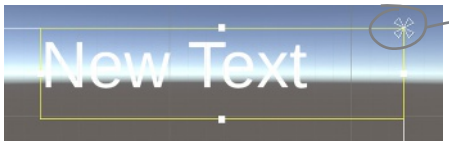
Since the Text only "lives" in the 2D Canvas, it uses a *Rect Transform* (it has that name because its position is relative to the rectangle that makes up the Canvas). Click on the Anchors box to display the anchor presets.



Make sure you hold down both Shift and Alt (Option on a Mac) so you set both the pivot and position.

The Anchor Presets window lets you anchor your UI GameObjects to various parts of the Canvas. **Hold down Alt and Shift** (or Option+Shift on a Mac) and **choose the top right anchor preset**. Click the same button you used to bring up the Anchor Presets window. The Text is now in the upper-right corner of the Canvas—double-click on it again to zoom into it.





You set the anchor pivot to the top right. The Text's position is the anchor's position relative to the Canvas.

Let's add a little space above and to the right of the Text. Go back to the Rect Transform panel and **set both Pos X and Pos Y to −10** to position the text 10 units to the left and 10 down from the top-right corner. Then **set the Alignment on the Text component to *right***, and use the box at the top of the Inspector to **change the GameObject's name to Score**.

Your new Text should now show up in the Hierarchy window with the name Score. It should now be right-aligned, with a small gap between the edge of the Text and the edge of the Canvas.

Add a Button that calls a method to start the game

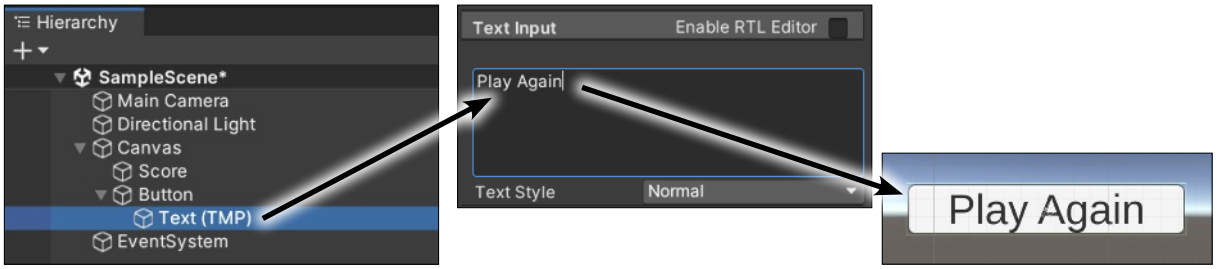
When the game is in its "game over" mode, it will display a button labeled Play Again that calls a method to restart the game. **Add an empty StartGame method** to your GameController class (we'll add its code later):

```
public void StartGame()
{
}
```

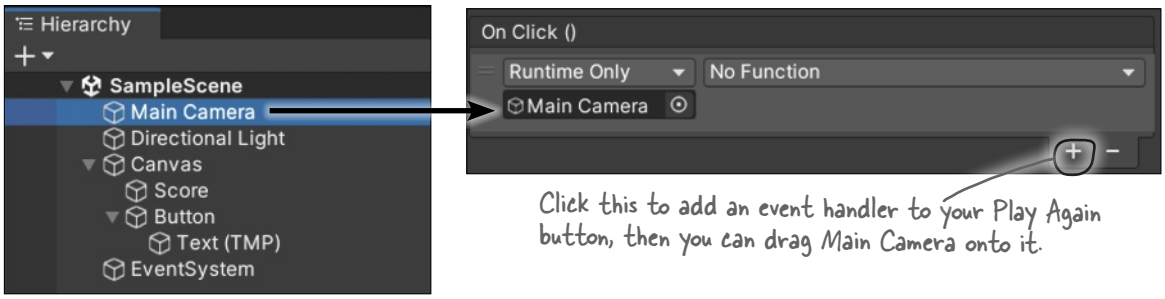
We'll add the code for this method later. For now, just leave it empty.

Click on Canvas in the Hierarchy window to focus on it. Then **choose UI >> Button - TextMeshPro** from the GameObject menu to add a Button. Use the context menu ( ) to **reset the Rect Transform** (which will move the button to the center of the Canvas) then **set its Width to 160 and Height to 30**.

There's a triangle next to Button in the Hierarchy—expand it. There's a Text (TMP) GameObject nested under it. Click on it and use the Inspector to set its text to `Play Again`.

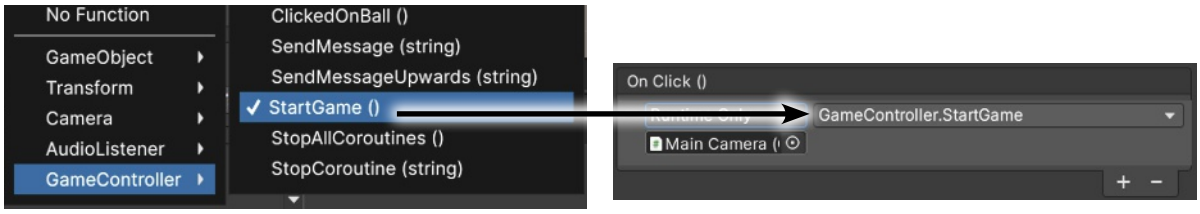


Now that the Button is set up, we just need to make it call the StartGame method on the GameController object attached to the Main Camera. A UI button is ***just a GameObject with a Button component***, and you can use its On Click () box in the Inspector to hook it up to an event handler method. Click the button at the bottom of the On Click () box to add an event handler, then **drag Main Camera onto the None (Object) box**.



Click this to add an event handler to your Play Again button, then you can drag Main Camera onto it.

Now the Button knows which GameObject to use for the event handler. Click the No Function dropdown and choose **GameController >> StartGame**. Now when the player presses the button, it will call the StartGame method on the GameController object hooked up to the Main Camera.



Make the Play Again button and Score Text work

Your game's UI will work like this:

- The game starts in the "game over" mode.
- Clicking the Play Again button starts the game.
- Text in the upper-right corner of the screen displays the current score.

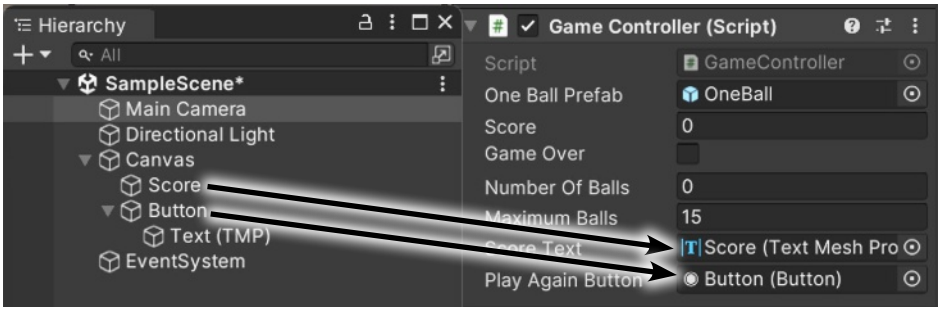
You'll be using the Text and Button classes in your code. They're in the UnityEngine.UI namespace, so **add these `using` directives** to the top of your GameController class:

```
using TMPro;
using UnityEngine.UI;
```

Now you can add Text and Button fields to your GameController (just above the OneBallPrefab field):

```
    public TMP_Text ScoreText;
    public Button PlayAgainButton;
```

Click on Main Camera in the Hierarchy window. **Drag the Text GameObject** out of the Hierarchy and ***onto*** the Score Text field in the Script component, **then drag the Button GameObject *onto*** the Play Again Button field.



Go back to your GameController code and **set the GameController field's default value to true**:

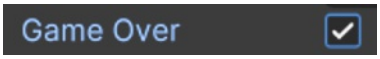
```
    public bool GameOver = true;
```

← Change this from false to true.

Now check the Script component in the Inspector in Unity.

Hold on, something's wrong!

The Unity editor still shows the Game Over checkbox as unchecked—it didn't change the field value. Make sure to check the box so your game starts in the "game over" mode:



Now the game will start in the "game over" mode, and the player can click the Play Again button to start playing.

Watch it!

Unity remembers your scripts' field values.

When you wanted to change the GameController. GameOver field from false to true, it wasn't enough to change the code. When you add a Script component to Unity, it keeps track of the field values, and it won't reload the default values unless you reset it from the context menu (⋮).

Finish the code for the game

The GameController object attached to the Main Camera keeps track of the score in its Score field. **Add an Update method to the GameController class** to update the Score Text in the UI:

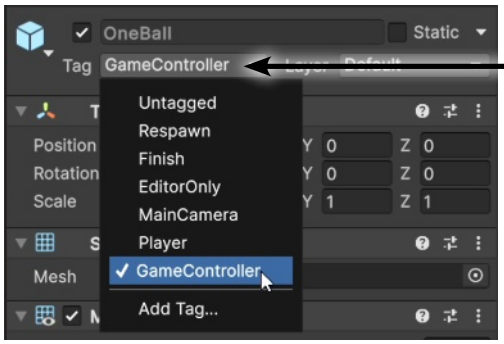
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

Next, **modify your GameController.AddABall method** to enable the Play Again button when it ends the game:

```
if (NumberOfBalls >= MaximumBalls)
{
    GameOver = true;
    PlayAgainButton.gameObject.SetActive(true);
}
```

Every GameObject has a property called gameObject that lets you manipulate it. You'll use its SetActive method to make the Play Again button visible or invisible.

There's just one more thing to do: get your StartGame method working so that it starts the game. It needs to do a few things: destroy any balls that are currently flying around the scene, disable the Play Again button, reset the score and number of balls, and set the mode to "running." You already know how to do most of those things! You just need to be able to find the balls in order to destroy them. **Click on the OneBall prefab in the Project window and set its tag**:



A *tag* is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them. When you click on a prefab in the Project window and use this dropdown to assign a tag, that tag will be assigned to every instance of that prefab that you instantiate.

Now you have everything in place to fill in your StartGame method. It uses a `foreach` loop to find and destroy any balls left over from the previous game, hides the button, resets the score and number of balls, and changes the game mode:

```
public void StartGame()
{
    foreach (GameObject ball in GameObject.FindGameObjectsWithTag("GameController"))
    {
        Destroy(ball);
    }
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    NumberOfBalls = 0;
    GameOver = false;
}
```

Do balls start appearing even if your game is in "game over" mode? Make sure the Game Over field is checked in the Main Camera's GameController script. You can reset it to its default value of true using the context menu.

Now run your game. It starts in "game over" mode. Press the button to start the game. The score goes up each time you click on a ball. As soon as the 15th ball is instantiated, the game ends and the Play Again button appears again.

Exercise

This exercise uses a lot of the C# knowledge you've built up over the last 8 chapters. Remember—if you get stuck, it's not cheating to peek at the solution!

Here's a Unity coding challenge for you! Each of your GameObjects has a **transform.Translate** method that moves it a distance from its current position. The goal of this exercise is to modify your game so that instead of using transform.RotateAround to circle balls around the Y axis, your OneBallBehaviour script uses transform.Translate to make the balls fly randomly around the scene.

Remove the XRotation, YRotation, and ZRotation fields from OneBallBehaviour. **Replace them with fields** to hold the X, Y, and Z speed called XSpeed, YSpeed, and ZSpeed. They're float fields—no need to set their values.

Replace all of the code in the Update method with this line of code that calls the transform.Translate method:

```
transform.Translate(Time.deltaTime * XSpeed,
                    Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);
```

The parameters represent the speed that the ball is traveling along the X, Y, or Z axis. So if XSpeed is 1.75, multiplying it by Time.deltaTime causes the ball to move along the X axis at a rate of 1.75 units per second.

Replace the DegreesPerSecond field with a field called Multiplier with a value of 0.75**F**—the **F** is important! Use it to update the XSpeed field in the Update method, and **add two similar lines** for the YSpeed and ZSpeed fields:

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

Part of this exercise is to **understand exactly how this line of code works**. Random.value is a static method that returns a random floating-point number between 0 and 1. What is this line of code doing to the XSpeed field?

.....

.....

.....

Then **add a method called ResetBall** and call it from the Start method. Add this line of code to ResetBall:

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

Before you start working on the game, figure out what these lines of code do.

.....

.....

Add two more lines just like it to ResetBall that update YSpeed and ZSpeed. Then **move the line of code** that updates transform.position out of the Start method and into the ResetBall method.

Modify the OneBallBehaviour class to **add a field called TooFar** and set it to 5. Then modify the Update method to check whether the ball went too far. You can check if a ball went too far along the X axis like this:

```
Mathf.Abs(transform.position.x) > TooFar
```

That checks the *absolute value* of the X position, which means that it will check if transform.position.x is greater than 5F or less than –5F. Here's an `if` statement that checks if the ball went too far along the X, Y, or Z axis:

```
if ((Mathf.Abs(transform.position.x) > TooFar)
     || (Mathf.Abs(transform.position.y) > TooFar)
     || (Mathf.Abs(transform.position.z) > TooFar)) {
```

Modify your OneBallBehaviour.Update method to use that `if` statement to call ResetBall if the ball went too far.

Exercise Solution

Here's what the entire OneBallBehaviour class looks like after updating it following the instructions in the exercise. The key to how this game works is that each ball's speed along the X, Y, and Z axes is determined by its current XSpeed, YSpeed, and ZSpeed values. By making small changes to those values, you've made your ball move randomly throughout the scene.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OneBallBehaviour : MonoBehaviour
{
    public float XSpeed;
    public float YSpeed;
    public float ZSpeed;
    public float Multiplier = 0.75F;
    public float TooFar = 5;

    static int BallCount = 0;
    public int BallNumber;

    // Start is called before the first frame update
    void Start()
    {
        BallCount++;
        BallNumber = BallCount;

        ResetBall();
    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(Time.deltaTime * XSpeed,
                            Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);

        XSpeed += Multiplier - Random.value * Multiplier * 2;
        YSpeed += Multiplier - Random.value * Multiplier * 2;
        ZSpeed += Multiplier - Random.value * Multiplier * 2;

        if ((Mathf.Abs(transform.position.x) > TooFar)
             || (Mathf.Abs(transform.position.y) > TooFar)
             || (Mathf.Abs(transform.position.z) > TooFar))
        {
            ResetBall();
        }
    }
```

You added these fields to your OneBallBehaviour class. Don't forget to add the F to 0.75F; otherwise, your code won't build.

When the ball is first instantiated, its Start method calls ResetBall to give it a random position and speed.

The Update method first moves the ball, then updates the speed, and finally checks if it went out of bounds. It's OK if you did these things in a different order.

Exercise Solution

```
    void ResetBall()
    {
        XSpeed = Multiplier - Random.value * Multiplier * 2;
        YSpeed = Multiplier - Random.value * Multiplier * 2;
        ZSpeed = Multiplier - Random.value * Multiplier * 2;

        transform.position = new Vector3(3 - Random.value * 6,
            3 - Random.value * 6, 3 - Random.value * 6);
    }

    void OnMouseDown()
    {
        GameController controller = Camera.main.GetComponent<GameController>();
        if (!controller.GameOver)
        {
            controller.ClickedOnBall();
            Destroy(gameObject);
        }
    }
}
```

We reset the ball when it's first instantiated or if it flies out of bounds by giving it a random speed and position. It's OK if you set the position first.

Here are our answers to the questions—did you come up with similar answers?

By increasing or decreasing the ball's speed along all three axes, we're giving each ball a wobbly random path.

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

What is this line of code doing to the XSpeed field?

Random.value * Multiplier * 2 finds a random number between 0 and 1.5. Subtracting that from Multiplier gives us a random number between –0.75 and 0.75. Adding that value to XSpeed causes it to either speed up or slow down a small amount for each frame.

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

It sets the XSpeed field to a random value between –0.75 and 0.75. This causes some balls to start going forward along the X axis and others to go backward, all at different speeds.

Did you notice that you *didn't have to make any changes to the GameController class*? That's because you didn't make changes to the things that GameController does, like managing the UI or the game mode. That's a great example of how separation of concerns can be really useful! If you can make a change by modifying one class but not touching others, that can be a sign that you designed your classes well.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas:

- Is the game too easy? Too hard? Try changing the parameters that you pass to InvokeRepeating in your GameController.Start method. Try making them fields. Play around with the MaximumBalls value too. Small changes in these values can make a big difference in gameplay.
- We gave you texture maps for all of the billiard balls. Try adding different balls that have different behaviors. Use the scale to make some balls bigger or smaller, and change their parameters to make them go faster or slower, or move differently.
- Can you figure out how to make a "shooting star" ball that flies off really quickly in one direction and is worth a lot if the player clicks on it? How about making a "sudden death" 8 ball that immediately ends the game?
- Modify your GameController.ClickedOnBall method to take a `score` parameter instead of incrementing the Score field and add the value that you pass. Try giving different values to different balls.

If you change fields in the OneBallBehaviour script, don't forget to* reset the Script component *of the OneBall prefab! Otherwise, it will remember the old values.

The more practice you get writing C# code, the easier it will get. Getting creative with your game is a great opportunity to get some practice!

Bullet Points

- Unity games display a **user interface (UI)** with controls and graphics on a flat, two-dimensional plane in front of the game's 3D scene.
- Unity provides a set of **2D UI GameObjects** specifically made for building user interfaces.
- A **Canvas** is a 2D GameObject that lets you lay out your game's UI. UI components like Text and Button are nested under a Canvas GameObject.
- The **2D button** at the top of the Scene window toggles 2D view on and off, which makes it easier to lay out a UI.
- When you add a **Script component** to Unity, it keeps track of the field values. You can reload the default values by resetting them from the context menu.
- A **Button** can call any method in a script that's attached to a GameObject.
- You can use the Inspector to **modify field values** in your GameObjects' scripts. If you modify them while the game is running, they'll reset to saved values when it stops.
- The **transform.Translate** method moves a GameObject a distance from its current position.
- A **tag** is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them.
- The GameObject.FindGameObjectsWithTag method returns a collection of GameObjects that match a given tag.

9 LINQ and lambdas

Get control of your data

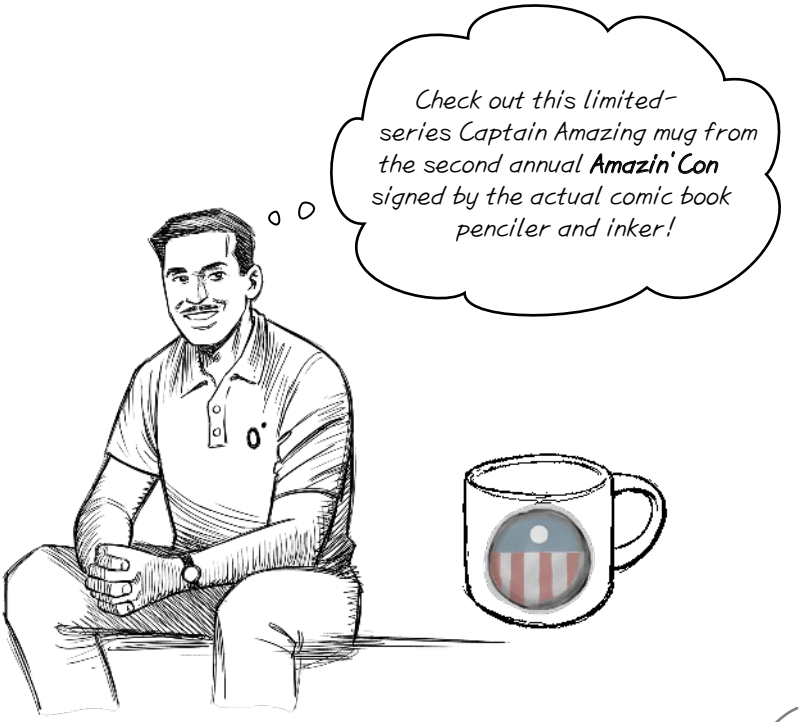
It's a data-driven world…we all need to know how to live in it.

Gone are the days when you could program for days, even weeks, without dealing with **loads of data**. Today, ***everything is about data***, and that's where **LINQ** comes in. LINQ is a feature of C# and .NET that not only lets you **query data** in your .NET collections in an intuitive way, but lets you **group data** and **merge data from different data sources**. You'll use **anonymous objects** to manage your data in new and interesting ways. You'll add **unit tests** to make sure your code is working the way you want. Once you've got the hang of wrangling your data into manageable chunks, you can use **lambda expressions** to refactor your C# code to make it easier to work with and more expressive.

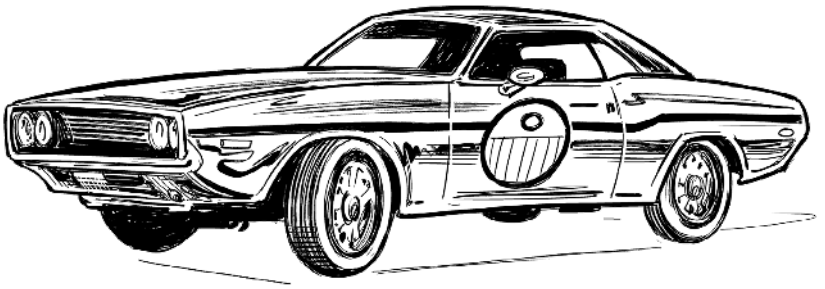
Jimmy's a Captain Amazing superfan...

Meet Jimmy, one of the most enthusiastic collectors of Captain Amazing comics, graphic novels, and paraphernalia. He knows all the Captain trivia, he's got props from all the movies, and he's got a comic collection that can only be described as, well, amazing.

Jimmy tracked down this rare 2005 Captain Amazing action figure in its original packaging.

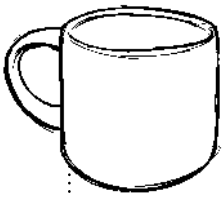


That's right, that's the actual Amazingmobile stunt car from the flop Captain Amazing TV show that ran from September through November 1973. How'd Jimmy even get his hands on it?



...but his collection's all over the place

Jimmy may be passionate, but he's not exactly organized. He's trying to keep track of the most prized "crown jewel" comics of his collection, but he needs help. Can you build Jimmy an app to manage his comics?



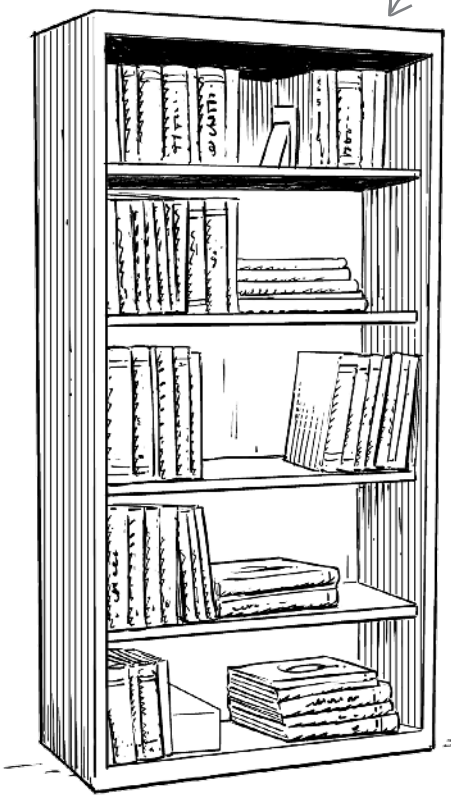
Relax

LINQ works with values *and* objects

We're going to use LINQ to query collections of numbers to introduce the ideas and syntax. You'll probably be thinking, "But how does this help manage comic books?" That's a great question to keep in mind in the first part of this chapter. Later on, we'll use really similar LINQ queries to manage collections of Comic objects.

Framed cover of the legendary "Death of the Object" issue, signed by the writers.

Jimmy has an impressive collection of classic Captain Amazing comics and graphic novels.

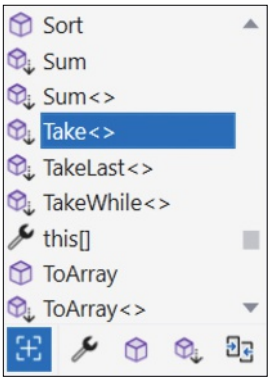


Use LINQ to query your collections

In this chapter, you'll learn about **LINQ** (or **Language Integrated Query**). LINQ combines really useful classes and methods with powerful features built directly into C#, all created to help you work with sequences of data—like Jimmy's comic book collection.

Let's use Visual Studio or VSCode to start exploring LINQ. **Create a new Console App** project and give it the name **LinqTest**. Add this code, and when you get to the last line, **add a period** and look at the IntelliSense window:

```
List<int> numbers = new List<int>();

for (int i = 1; i <= 99; i++) {
    numbers.Add(i);
}

IEnumerable<int> firstAndLastFive = numbers.
```



You've been using arrays and collections for most of the book. Have you noticed that there are a LOT of methods in the IntelliSense dropdown for them?

Most of these methods show up for arrays, Lists, Dictionaries, Queues, and Stacks. Why do you think they all share so many methods?

Use the IntelliSense window to see the descriptions of the Take, Concat, and TakeLast methods.

```
(extension) IEnumerable<int> IEnumerable<int>.Take<int>(int count) (+ 1 generic overload)
Returns a specified number of contiguous elements from the start of a sequence.
```

```
(extension) IEnumerable<int> IEnumerable<int>.Concat<int>(IEnumerable<int> second)
Concatenates two sequences.
```

```
(extension) IEnumerable<int> IEnumerable<int>.TakeLast<int>(int count)
Returns a new enumerable collection that contains the last count elements from source.
```

Let's use those new methods to finish your console app:

```
IEnumerable<int> firstAndLastFive = numbers.Take(5).Concat(numbers.TakeLast(5));
foreach (int i in firstAndLastFive)
{
    Console.Write($"{i} ");
}
```

Now run your app. It prints this line of text to the console:

```
1 2 3 4 5 95 96 97 98 99
```

So what did you just do?

LINQ (Language INtegrated Query) is a combination of C# features and .NET classes that helps you work with sequences of data.

A LINQ Query Up Close

Let's take a closer look at how you're using the LINQ methods Take, TakeLast, and Concat.

```
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

`numbers`

This is the original List<int> that you created with a **`for`** loop.

`numbers.Take(5)`

The Take method takes the first elements from a sequence.

```
1 2 3 4 5
```

`numbers.TakeLast(5)`

The TakeLast method takes the last elements from a sequence.

```
96 97 98 99 100
```

`numbers.Take(5).Concat(numbers.TakeLast(5))`

The Concat method concatenates two sequences together.

```
1 2 3 4 5 96 97 98 99 100
```

Use method chaining with LINQ methods

The Take, Concat, TakeLast, and many other LINQ methods are available on arrays, queues, stacks...in fact, you can use them with any object that extends IEnumerable<T>. Since almost all of the LINQ methods return an IEnumerable<T>, you can take the results of a LINQ method and call another LINQ method directly on them without using a variable to keep track of the results. That's called method chaining, and it lets you write very compact code.

For example, we could have used variables to store the results of Take and TakeLast:

```
IEnumerable<int> firstFive = numbers.Take(5);
IEnumerable<int> lastFive = numbers.TakeLast(5);
IEnumerable<int> firstAndLastFive = firstFive.Concat(lastFive);
```

But method chaining lets us put it all into a single line of code, calling Concat directly on the output of numbers.Take(5). They both do the same thing. Keep in mind that compact code is not necessarily better than verbose code! Sometimes breaking a chained method call into extra lines of code makes it clearer and easier to understand. It's up to you to decide which is more readable for any particular project.

LINQ works with any sequence

In Chapter 7, we learned that interfaces define specific jobs that classes and objects do, and IEnumerable<T> is no exception. When an object implements IEnumerable<T>, any instance of that class does a specific job: it can be a **sequence**, or a bunch of values, objects, or other elements that come one after another.

- The list of numbers from 1 to 99 was a sequence.
- When you called its Take method, it returned a reference to a sequence that contained the first five elements.
- When you called its TakeLast method, it returned another five-element sequence.
- And when you used Concat to combine the two five-element sequences, it created a new sequence with 10 elements and returned a reference to that new sequence.

Any time you have an object that implements the IEnumerable interface, you have a sequence that you can use with LINQ. Doing an operation on that sequence in order is called enumerating the sequence.

LINQ methods enumerate your sequences—just like foreach loops

You already know that `foreach` loops work with IEnumerable objects. Think about what a `foreach` loop does:

This foreach loop operates on the sequence 1, 2, 3, 4, 5, 96, 97, 98, 99, 100.

```
foreach (int i in firstAndLastFive)
{
    Console.Write($"{i} ");
}
```

It starts at the first element of the sequence (in this case, 1)...

...and it does an operation on each element in that sequence in order (writing a string to the console).

When a method goes through each item in a sequence in order, that's called **enumerating** the sequence. And that's how LINQ methods work.

When an object implements the IEnumerable<T> interface, it's a *sequence* that can be *enumerated*. That means it represents series of values, objects, or other elements in a specific order.

That's the job that an object implementing the IEnumerable interface does.

e-nu-mer-ate, verb.
mention a number of things one by one. *Suzy* ***enumerated*** *the toy cars in her collection for her dad, telling him each car's make and model.*

```
Enumerable.Range(8, 5);
```

```
8 9 10 11 12
```

What if you want to find the first 30 issues in Jimmy's collection starting with issue #118? LINQ provides a really useful method to help with that. The static *Enumerable.Range method* generates a sequence of integers. Calling Enumerable.Range(8, 5) returns a sequence of 5 numbers starting with 8: 8, 9, 10, 11, 12.

IEnumerable<int> Enumerable.Range(int start, int count)
Generates a sequence of integral numbers within a specified range.

Exceptions:
ArgumentOutOfRangeException

You'll get some practice with the Enumerable.Range method in the next pencil-and-paper exercise.

Sharpen your pencil

Here are just a few of the LINQ methods that you'll find on any sequence that implements IEnumerable<T>. Most of the LINQ methods have pretty intuitive names. Can you figure out just from their names what they do? **Draw a line** connecting each method call to its output.

```
Enumerable.Range(1, 5)
    .Sum()
```

```
Enumerable.Range(1, 6)
    .Average()
```

```
new int[] { 3, 7, 9, 1, 10, 2, -3 }
    .Min()
```

```
new int[] { 8, 6, 7, 5, 3, 0, 9 }
    .Max()
```

```
Enumerable.Range(10, 3721)
    .Count()
```

```
Enumerable.Range(5, 100)
    .Last()
```

```
new List<int>() { 3, 8, 7, 6, 9, 6, 2 }
    .Skip(4)
    .Sum()
```

```
Enumerable.Range(10, 731)
    .Reverse()
    .Last()
```

```
9
```

```
17
```

```
104
```

```
15
```

```
3.5
```

```
10
```

```
-3
```

```
3721
```

Sharpen your pencil Solution

The Sum method adds up all the values in the sequence and returns their sum.

```
Enumerable.Range(1, 5)
    .Sum()
```

```
Enumerable.Range(1, 6)
    .Average()
```

```
new int[] { 3, 7, 9, 1, 10, 2, -3 }
    .Min()
```

```
new int[] { 8, 6, 7, 5, 3, 0, 9 }
    .Max()
```

```
Enumerable.Range(10, 3721)
    .Count()
```

```
Enumerable.Range(5, 100)
    .Last()
```

```
new List<int>() { 3, 8, 7, 6, 9, 6, 2 }
    .Skip(4)
    .Sum()
```

```
Enumerable.Range(10, 731)
    .Reverse()
    .Last()
```

9

Skip(4) skips the first four elements in the sequence, returning {6, 9, 2}. Sum adds them up: 6 + 9 + 2 = 17.

17

104

Range(5, 100) returns {5, 6, 7, ..., 102, 103, 104}, and Last() gets the last number in that sequence.

15

3.5

10

Range(10, 731) returned a sequence of 731 numbers starting with 10. Reverse does exactly what it sounds like—it reverses the order—so the last element in the reversed sequence is 10.

-3

3721

The LINQ methods in this exercise have names that make it obvious what they do. Some LINQ methods, like Sum, Min, Max, Count, First, and Last, return a single value. The Sum method adds up the values in the sequence. The *Average* method returns their average value. The *Min* and *Max* methods return the smallest and largest values in the sequence. The *First* and *Last* methods do exactly what it sounds like they do.

Other LINQ methods, like *Take*, *TakeLast*, *Concat*, *Reverse* (which reverses the order in a sequence), and *Skip* (which skips the first elements in a sequence), return another sequence.

LINQ's query syntax

The LINQ methods you've seen so far may not be enough on their own to answer the kinds of questions about data that we might have—or the questions that Jimmy has about his comic collection.

And that's where the **LINQ declarative query syntax** comes in. It uses special keywords—including `where`, `select`, `groupby`, and `join`—to build **queries** directly into your code.

LINQ queries work on *sequences*, or objects that implement IEnumerable<T>. LINQ queries start with a `from` clause:

```
from (variable) in (sequence)
```

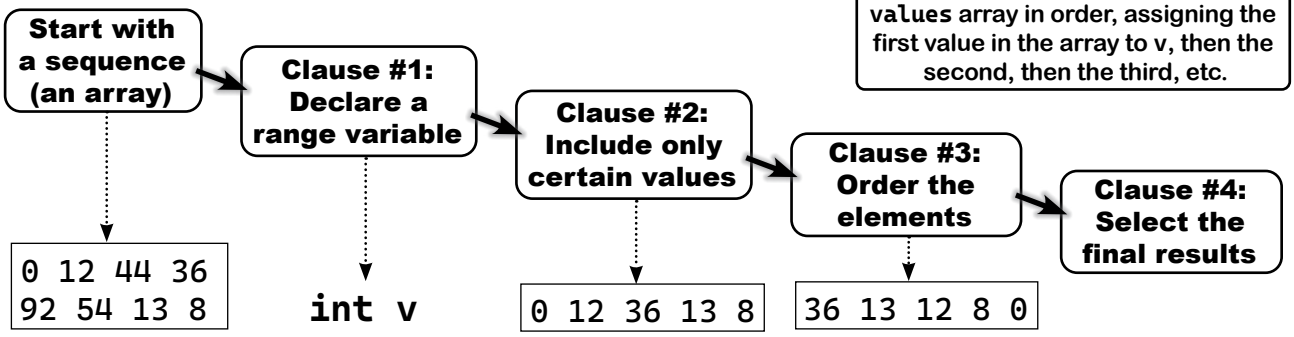
It tells the query what sequence to execute against, and assigns a name to use for each element being queried. It's like the first line in a `foreach` loop: it declares a variable to use while iterating through the sequence that's assigned each element in the sequence. So this:

```
from v in values
```

iterates through each element in the `values` array in order, assigning the first value in the array to `v`, then the second, then the third, etc.

LINQ queries are built from clauses

Let's build a query that finds the numbers in an int array that are under 37 and puts those numbers in ascending order. It does that using four **clauses** that tell it what object to query, how to determine which of its members to select, how to sort the results, and how the results should be returned.




```
int[] values = [0, 12, 44, 36, 94, 54, 13, 8];
IEnumerable<int> result =
  from v in values
  where v < 37
  orderby -v
  select v;
```

This LINQ query has four clauses: the from clause, a where clause, an orderby clause, and the select clause.

The *from clause* assigns a variable, called the *range variable*, to stand in for each value as it iterates through the array. In the first iteration the variable v is 0, then 12, then 44, etc.

A *where clause* contains a conditional test that the query uses to determine which values to include in the results—in this case, any value less than 37.

An *orderby clause* contains an expression used to sort the results—in this case, -v sorts from highest to lowest.

```
// use a foreach loop to print the results
foreach(int i in result)
  Console.Write($"{i} ");
```

The query ends with a *select clause* that tells it what to put in the results.

Output: 36 13 12 8 0

there are no Dumb Questions

Q: I'm still not clear on what method chaining is. How does it work, exactly, and why should I use it?

A: Method chaining is a really common way of making multiple method calls in a row. Since many of the LINQ methods return a sequence that implements IEnumerable<T>, you can call another LINQ method on the result. Method chaining isn't unique to LINQ either. You can use it in your own classes.

Q: Can you show an example of method chaining?

A: Sure. Let's add a method to the Guy class that uses method chaining to change the Name property name. We'll add a Rename method that ***returns an instance of the Guy class***. We'll just use the `this` keyword to return that same instance of the object.

```
public Guy Rename(string newName) {
    this.Name = newName;
    return this;
}
```

Now you can create a new Guy, then chain a call to Rename:

```
Guy guy = new Guy("Harold", 37)
          .Rename("Harry");
```

This is a really good question to ask your favorite AI chatbot. Ask it to give you an example of a C# class that uses method chaining, then keep asking it this same question over and over again.

Q: I think I get it. But just so I'm 100% clear, can you show me another example of a class that uses method chaining?

A: Sure. Here's a class that has two methods built for chaining:

```
class AddSubtract
{
  public int Value { get; set; }
  public AddSubtract Add(int i) {
      Console.WriteLine($"Value={Value}, adding {i}");
      return new AddSubtract() {
           Value = Value + i
      };
  }

  public AddSubtract Subtract(int i) {
      Console.WriteLine(
          $"Value={Value}, subtracting {i}");
      return new AddSubtract() { Value = Value - i };
  }
}
```

AddSubtract's Add and Subtract methods return AddSubtract instances.

You can call it like this:

```
AddSubtract a = new AddSubtract() { Value = 5 }
  .Add(5)
  .Subtract(3)
  .Add(9)
  .Subtract(12);
Console.WriteLine($"Result: {a.Value}");
```

Try adding the AddSubtract class to a new console app, then add these top-level statements.

This all sounds fine. But how does it help me manage my unruly comic book collection?

LINQ isn't just for numbers. It works with objects too.

When Jimmy looks at stacks and stacks of disorganized comics, he might see paper, ink, and a jumbled mess. When we developers look at them, we see something else: **lots and lots of data** just waiting to be organized. How do we organize comic book data in C#? The same way we organize playing cards, bees, or items on Sloppy Joe's menu: we create a class, then we use a collection to manage that class. So all we need to help Jimmy out is a Comic class, and code to help us bring some sanity to his collection. And LINQ will help!

LINQ works with objects

Do this!

Jimmy wanted to know how much some of his prize comics are worth, so he hired a professional comic book appraiser to give him prices for each of them. It turns out that some of his comics are worth a lot of money! Let's use collections to manage that data for him.

1 **Create a new console app and add a Comic class.**

Use a primary constructor with name and issue arguments, and properties that return them:

```
class Comic(string name, int issue)
{
    public string Name { get { return name;} }
    public int Issue { get { return issue;} }

    public override string ToString() => $"{Name} (Issue #{Issue})";
```

What do you think that => does? Can you figure it out from the context?

We haven't used that => operator before! Can you figure out from its context what it does? You know how ToString methods work—so somehow the => makes the ToString return the interpolated string to the right of the operator.

2 **Add a List that contains Jimmy's catalog.**

Add this static Catalog field to the Comic class. It returns a sequence with Jimmy's prized comics:

```
    public static readonly IEnumerable<Comic> Catalog = [
          new Comic("Johnny America vs. the Pinko", 6),
          new Comic("Rock and Roll (limited edition)", 19),
          new Comic("Woman's Work", 36),
          new Comic("Hippie Madness (misprinted)", 57),
          new Comic("Revenge of the New Wave Freak", 68),
          new Comic("Black Monday", 74),
          new Comic("Beyond the Alternative", 83),
          new Comic("The Death of the Object", 97),
    ];
```

3 **Use a Dictionary to manage the prices.**

Add a static Comic.Prices field—it's a Dictionary<int, decimal> that lets you look up the price of each comic using its issue number (using the collection initializer syntax for dictionaries that you learned in Chapter 8). Note that we're using the **IReadOnlyDictionary interface** for encapsulation—it's an interface that includes only the methods to read values (so we don't accidentally change the prices):

```
    public static readonly IReadOnlyDictionary<int, decimal> Prices =
        new Dictionary<int, decimal>() {
            { 6, 3600M },
            { 19, 500M },
            { 36, 650M },
            { 57, 13525M },
            { 68, 250M },
            { 74, 75M },
            { 83, 25.75M },
            { 97, 35.25M },
        };
    }
```

We're using the IReadOnlyDictionary interface to make the Dictionary read-only, which will prevent other classes from changing prices.

Jimmy's rare misprinted edition of issue #57 ("Hippie Madness" from 1973) is worth 13,525 bucks. Wow!

We used a Dictionary to store the prices for the comics. We could have included a property called Price. We decided to keep information about the comic and price separate. We did this because prices for collectors' items change all the time, but the name and issue number will always be the same. *Do you think we made the right choice?*

Use a LINQ query to finish the app for Jimmy

We used the LINQ declarative query syntax earlier to create a query with four clauses: a `from` clause to create a range variable, a `where` clause to include only numbers under 37, an `orderby` clause to sort them in descending order, and a `select` clause to determine which elements to include in the resulting sequence.

Let's **add a LINQ query to the top-level statements**. The query works exactly the same way as the one you saw earlier, except that ***it uses Comic objects instead of int values***, so it writes a list of comics with a price > 500 in reverse order to the console. We'll start with two `using` declarations so we can use IEnumerable<T> and LINQ methods. The query will return an IEnumerable<Comic>, then use a `foreach` loop to iterate through it and write the output.

4 **Modify the top-level statements method to use a LINQ query.**

Here are top-level statements that run a LINQ query and use a `foreach` loop to print the results:

```
IEnumerable<Comic> mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby -Comic.Prices[comic.Issue]
    select comic;

foreach (Comic comic in mostExpensive)
{
    Console.WriteLine($"{comic} is worth {Comic.Prices[comic.Issue]:c}");
}
```

Keep your eye on how the "comic" range variable is used. It's a Comic variable that's declared in the from clause and used in the where and orderby clauses.

The select clause determines what the query returns. Since it's selecting a Comic variable, the result of the query is an IEnumerable<Comic>.

We saw in previous chapters that ":c" formats a number as local currency, so if you're in the UK, you'll see £ instead of $ in the output.

Output:

```
Hippie Madness (misprinted) is worth $13,525.00
Johnny America vs. the Pinko is worth $3,600.00
Woman's Work is worth $650.00
```

5 **Use the `descending` keyword to make your `orderby` clause more readable.**

Your `orderby` clause uses a minus sign to negate the comic prices before sorting, causing the query to sort them in descending order. But it's easy to accidentally miss that minus sign when you're reading the code and trying to figure out how it works. Luckily, there's another way to get the same results. **Remove the minus sign and add the `descending` keyword** to the end of the clause:

```
IEnumerable<Comic> mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby Comic.Prices[comic.Issue] descending
    select comic;
```

The descending keyword makes orderby sort in reverse order.

Anatomy of a Query

Let's explore how LINQ works by making a couple of small changes to the query:

- That minus in the **`orderby`** clause is easy to miss. We'll make sure to use the **`descending`** keyword that you just added in step **5**.
- The **`select`** clause you just wrote selected the comic, so the result of the query was a sequence of Comic references. Let's replace it with an interpolated string that uses the **`comic`** range variable—now the result of the query is a sequence of strings.

Changing the select clause causes the query to return a sequence of strings.

Here's the updated LINQ query. Each clause in the query produces a sequence that feeds into the next clause—we've added a table under each clause that shows its result.

```
IEnumerable<string> mostExpensiveComicDescriptions =
    from comic in Comic.Catalog
```

The from clause loops through Comic.Catalog, pulling out each value in it and assigning it to the range variable "comic". The result of the from clause is a sequence of Comic object references.

We'll show you the sequence that's generated by each clause in the query.

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Rock and Roll (limited edition)", Issue = 19 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Revenge of the New Wave Freak", Issue = 68 }
{ Name = "Black Monday", Issue = 74 }
{ Name = "Beyond the Alternative", Issue = 83 }
{ Name = "The Death of the Object", Issue = 97 }

```
    where Comic.Prices[comic.Issue] > 500
```

The where clause starts with the results of the from clause, assigning "comic" to each value and using it to apply a conditional test that checks the Comic.Prices Dictionary for the price and only includes comics whose price is greater than 500.

They query gets the price from the Dictionary.

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }

```
    orderby Comic.Prices[comic.Issue] descending
```

The orderby clause starts with the results of the where clause and sorts in descending order by price.

{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }

```
    select $"{comic} is worth {Comic.Prices[comic.Issue]:c}";
```

"Hippie Madness (misprinted) is worth $13,525.00"
"Johnny America vs. the Pinko is worth $3,600.00"
"Woman's Work is worth $650.00"

The select clause loops through the results of the orderby clause, using the "comic" range variable with string interpolation to return a sequence of strings.

The var keyword lets C# figure out variable types for you

We just saw that when we made the small change to the `select` clause, the type of sequence that the query returned changed. When it was select comic, the return type was IEnumerable<Comic>. When we changed it to `select $"{comic} is worth {Comic.Prices[comic.Issue]:c}";` the return type changed to IEnumerable<string>. When you're working with LINQ, that happens all the time—you're constantly tweaking your queries. It's not always obvious exactly what type they return. Sometimes going back and updating all of your declarations can get annoying.

Luckily, C# gives us a really useful tool to help keep variable declarations simple and readable. You can replace any variable declaration with the **`var` keyword**. So you can replace any of these declarations:

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
string s = $"The count is {numbers.Count()}";
IEnumerable<Comic> comics = new List<Comic>();
IReadOnlyDictionary<int, decimal> prices = Comic.Prices;
```

with these declarations, which do exactly the same thing:

```
var numbers = Enumerable.Range(1, 10);
var s = $"The count is {numbers.Count()}";
var comics = new List<Comic>();
var prices = Comic.Prices;
```

When you use the `var` keyword, you're telling C# to use an *implicitly typed* variable. We saw that same word—"implicit"—back in Chapter 8 when we talked about covariance. It means that C# figures out the types on its own.

And you don't have to change any of your code. Just replace the types with `var` and everything works.

When you use var, C# figures out the variable's type automatically

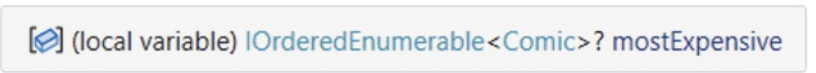
Go ahead—try it right now. Comment out the first line of the LINQ query you just wrote, then replace IEnumerable<Comic> with var:

```
// IEnumerable<Comic> mostExpensive =
var mostExpensive =
      from comic in Comic.Catalog
      where Comic.Prices[comic.Issue] > 500
      orderby -Comic.Prices[comic.Issue]
      select comic;
```

When you used "var" in the variable declaration, the IDE figured out its type based on how it was used in the code.

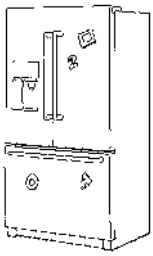
If you temporarily comment out the orderby clause in the query, that turns mostExpensive into an IEnumerable<T>.

Now hover your mouse cursor over the variable name `mostExpensive` to see its type:



The IDE figured out the `mostExpensive` variable's type—and it's a type we haven't seen before. C# automatically figured out that the `mostExpensive` variable is an IOrderedEnumerable<Comic>?—and we've seen that question mark before, any time a variable can potentially be set to null.

Remember in Chapter 7 when we talked about how interfaces can extend other interfaces? The IOrderedEnumerable interface is part of LINQ—it's used to represent a *sorted* sequence—and it extends the IEnumerable<T> interface. Try commenting out the `orderby` clause and hovering over the `mostExpensive` variable—you'll find that it turns into an IEnumerable<Comic>. That's because C# looks at the code to ***figure out the type of any variable you declare with*** `var`.



LINQ Magnets

We had a nice LINQ query that used the `var` keyword arranged with magnets on the refrigerator—but someone slammed the door and the magnets fell off! Rearrange the magnets so they produce the output at the bottom of the page.

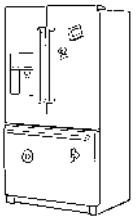
Two magnets stayed on the fridge! This must be the sequence of numbers you're starting with.

```
int[] badgers =
[ 36, 5, 91, 3, 41, 69, 8 ];
```

```
orderby
select
from
select
skunks
from
where
pigeon + 5;
weasels.Sum()
(pigeon != 36 && pigeon < 50)
pigeon descending
.Take(3);
sparrow - 1;
sparrow in bears
pigeon in badgers
var skunks =
var bears =
var weasels =
);
Console.WriteLine("Get your kicks on route {0}",
```

Output:

```
Get your kicks on route 66
```



LINQ Magnets Solution

Rearrange the magnets so they produce the output at the bottom of the page.

LINQ starts with some sort of sequence, collection, or array—in this case, an array of integers.

```
int[] badgers =
[ 36, 5, 91, 3, 41, 69, 8 ];
```

We chose confusing names like skunks, badgers, and bears on purpose. "from pigeon in badgers" makes for a good puzzle, but the code is pretty hard to understand. Renaming the array to "numbers" and using "from number in numbers" to declare a more sensibly named range variable would make it more readable.

```
var skunks =
    from pigeon in badgers
    where (pigeon != 36 && pigeon < 50)
    orderby pigeon descending
    select pigeon + 5;
```

After this statement, skunks contains four numbers: 46, 13, 10, and 8.

This LINQ statement pulls all the numbers that are below 50 and not equal to 36 out of the array, adds 5 to each of them, sorts them from biggest to smallest, puts them in a new object, and points the skunks reference at it.

```
var bears =
    skunks
    .Take(3);
```

After this statement, bears contains three numbers: 46, 13, and 10.

Here's where we take the first three numbers in skunks and put them into a new sequence called bears.

```
var weasels =
    from sparrow in bears
    select sparrow - 1;
```

After this statement, weasels contains three numbers: 45, 12, and 9.

This statement just subtracts 1 from each number in bears and puts them all into weasels.

```
Console.WriteLine("Get your kicks on route {0}",
    weasels.Sum() );
```

The numbers in weasels add up to 66.

45 + 12 + 9 = 66

Output:

```
Get your kicks on route 66
```

*Are you seriously telling me that I can replace the type in any variable declaration with **var** and my code will still work? It can't be that simple.*

You really can use `var` in your variable declarations.

And yes, it really is that simple. A lot of C# developers declare local variables using **`var`** almost all the time, and include the type only when it makes the code easier to read. As long as you're declaring the variable and initializing it in the same statement, you can use **`var`**.

But there are some important restrictions on using **`var`**. For example:

- You can only declare one variable at a time with **`var`**.
- You can't use the variable you're declaring in the declaration.
- You can't declare it equal to **`null`**.

If you create a variable named **`var`**, you won't be able to use it as a keyword anymore:

- You definitely can't use **`var`** to declare a field or a property—you can only use it as a local variable inside a method.
- If you stick to those ground rules, you can use **`var`** pretty much anywhere.

So when you did this in Chapter 4:

```
int hours = 24;
short RPM = 33;
int radius = 3;
char initial = 'S';
int balance = 345667 - 567;
```

If you wanted to define this as a long, you could replace 3 with 3L.

You could have done this:

```
var hours = 24;
var RPM = 33;
var radius = 3;
var initial = 'S';
var balance = 345667 - 567;
```

Or this in Chapter 6:

```
SwordDamage swordDamage = new SwordDamage(RollDice(3));
ArrowDamage arrowDamage = new ArrowDamage(RollDice(1));
```

Or this:

```
var swordDamage = new SwordDamage(RollDice(3));
var arrowDamage = new ArrowDamage(RollDice(1));
```

Or this in Chapter 8:

```
List<Card> cards = new List<Card>();
```

Or this:

```
var cards = new List<Card>();
```

...and your code would have worked exactly the same.

But you *can't* use `var` to declare a field or property:

```
class Card
{
    public var Value { get; set; }                          // If you try to use
    public var Suit { get; set; }                           // var like this, you'll
    public var Name { get { return $"{Value} of {Suit}"; } } // get compiler errors!
    ...
```

This won't work!

there are no Dumb Questions

Q: How does the `from` clause work?

A: It's a lot like the first line of a `foreach` loop. One thing that makes thinking about LINQ queries a little tricky is that you're not just doing one operation. A LINQ query does the same thing over and over again for each item in a collection—in other words, it enumerates the sequence. So the `from` clause does two things: it tells LINQ which collection to use for the query, and it assigns a name to use for each member of the collection that's being queried.

The way the `from` clause creates a new name for each item in the collection is really similar to how a `foreach` loop does it. Here's the first line of a `foreach` loop:

```
foreach (int i in values)
```

That `foreach` loop temporarily creates a variable called `i`, which it assigns sequentially to each item in the values collection. Now look at a `from` clause in a LINQ query on the same collection:

```
from i in values
```

That clause does pretty much the same thing. It creates a range variable called `i` and assigns it sequentially to each item in the values collection. The `foreach` loop runs the same block of code for each item in the collection, while the LINQ query applies the same criteria in the `where` clause to each item in the collection to determine whether or not to include it in the results.

Q: You took a LINQ query that returned a sequence of Comic references and made it return strings. What exactly did you do to make that work?

A: We modified the `select` clause. The `select` clause includes an expression that gets applied to every item in the sequence, and that expression determines the type of the output. So if your query produces a sequence of values or object references, you can use string interpolation in the `select` clause to turn each item in that sequence into a string. The query in the exercise solution ended with `select comic`, so it returned a sequence of Comic references. In our "Anatomy of a Query" code, we replaced it with `select $"{comic} is worth {Comic.Prices[comic.Issue]:c}"`—which caused the query to return a sequence of strings instead.

Q: How does LINQ decide what goes into the results?

A: That's what the `select` clause is for. Every LINQ query returns a sequence, and every item in that sequence is of the same type. It tells LINQ exactly what that sequence should contain. When you're querying an array or list of a single type—like an array of ints or a List<string>—it's obvious what goes into the `select` clause. What if you're selecting from a list of Comic objects? You could do what Jimmy did and select the whole class. You could also change the last line of the query to `select comic.Name` to tell it to return a sequence of strings. Or you could do `select comic.Issue` and have it return a sequence of ints.

Q: I understand how to use `var` in my code, but how does it actually work?

A: `var` is a keyword that tells the compiler to figure out the type of a variable at compilation time. The C# compiler figures out the type of the local variable that you're using LINQ to query. When you build your solution, the compiler will replace `var` with the right type for the data you're working with.

So when this line is compiled:

```
var result = from v in values
```

The compiler replaces `var` with this:

```
IEnumerable<int>
```

And you can always check a variable's type by hovering over it in the IDE.

The from clause in a LINQ query does two things: it tells LINQ which collection to use for the query, and it assigns a name to use for each member of the collection being queried.

Q: LINQ queries use a lot of keywords I haven't seen before—`from`, `where`, `orderby`, `select`…it's like a whole different language. Why does it look so different from the rest of C#?

A: Because LINQ serves a different purpose. Most of the C# syntax was built to do one small operation or calculation at a time. You can start a loop, set a variable, do a mathematical operation, call a method…those are all single operations. For example:

```
var under10 =
 from number in sequenceOfNumbers
 where number < 10
 select number;
```

That query looks pretty simple—not a lot of stuff there, right? But this is actually a pretty complex piece of code.

Think about what's got to happen for the program to actually select all the numbers from sequenceOfNumbers (which must be a reference to an object that implements IEnumerable<T>) that are less than 10. First, it needs to loop through the entire array. Then, each number is compared to 10. Then those results need to be gathered together so the rest of the code can use them.

And that's why LINQ looks a little odd: because it lets you cram a whole lot of behavior into a small—but easily readable!—amount of C# code.

Bullet Points

- When a class implements IEnumerable<T>, any instance of that class is a **sequence**, or a series of values, objects, or other elements in a specific order.
- When a method goes through each item in a sequence in order, that's called **enumerating** the sequence, which is how LINQ methods work.
- The **Take method** takes the first elements from a sequence. The **TakeLast method** takes the last elements from a sequence. The **Concat method** concatenates two sequences together.
- The **Average method** returns the average value of a sequence of numbers. The **Min and Max methods** return the smallest and largest values in the sequence.
- The **First and Last methods** return the first or last element in a sequence. The **Skip method** skips the first elements of a sequence and returns the rest.
- Many LINQ methods return a sequence, which lets you do **method chaining**, or calling another LINQ method directly on the results without using an extra variable to keep track of those results.
- The **IReadOnlyDictionary interface** is useful for encapsulation. You can assign any Dictionary to it to create a reference that doesn't allow the Dictionary to be updated.
- The **LINQ declarative query syntax** uses special keywords—including `where`, `select`, `groupby`, and `join`—to build queries directly into your code.
- LINQ queries start with a **`from` clause**, which assigns a variable to stand in for each value as it enumerates the sequence.
- The variable declared in the `from` clause is the **range variable**, and it can be used throughout the query.
- A **`where` clause** contains a conditional test that the query uses to determine which values to include in the results.
- An **`orderby` clause** contains an expression used to sort the results. You can optionally specify the **`descending`** keyword to reverse the sort order.
- The query ends with a **`select` clause** that includes an expression to specify what to include in the results.
- The **`var` keyword** is used to declare an implicitly typed variable, which means the C# compiler figures out the type of the variable on its own.
- You can use **`var` in place of a variable type** in any declaration statement that initializes a variable.
- You can include **a C# expression in a `select` clause** that gets applied to every element in the results, and determines the type of sequence returned by the query.

LINQ is versatile

With LINQ, you can do a lot more than just pull a few items out of a collection. You can modify the items before you return them. Once you've generated a set of result sequences, LINQ gives you a bunch of methods that work with them. Top to bottom, LINQ gives you the tools you need to manage your data. Let's do a quick review of some of the LINQ features that we've already seen.

You can use the `var` keyword to declare an *implicitly typed array*. Just use `new[]` along with a collection initializer, and the C# compiler will figure out the type of the array for you. If you mix and match types, you need to specify the array type:

```
var mixed = new object[] {
    1,
    "x",
    new Card(Values.Three, Suits.Clubs)
};
```

Modify every item returned from the query.

This code will add a string to the end of each element in an array of strings. It doesn't change the array itself—it creates a new sequence of modified strings.

```
var sandwiches = new[] { "ham and cheese", "salami with mayo",
                         "turkey and swiss", "chicken cutlet" };
var sandwichesOnRye =
    from sandwich in sandwiches
    select $"{sandwich} on rye";

foreach (var sandwich in sandwichesOnRye)
    Console.WriteLine(sandwich);
```

You can think of "select" as a way to make the same change to every element in a sequence—in this case, add "on rye" to the end.

Now all the items returned have "on rye" added to the end.

Output:

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

Perform calculations on sequences.

You can use the LINQ methods on their own to get statistics about a sequence of numbers.

```
var random = new Random();
var numbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    numbers.Add(random.Next(100));

Console.WriteLine($@"Stats for these {numbers.Count()} numbers:
The first 5 numbers: {String.Join(", ", numbers.Take(5))}
The last 5 numbers: {String.Join(", ", numbers.TakeLast(5))}
The first is {numbers.First()} and the last is {numbers.Last()}
The smallest is {numbers.Min()}, and the biggest is {numbers.Max()}
The sum is {numbers.Sum()}
The average is {numbers.Average():F2}");
```

The static *String.Join* method concatenates all of the items in a sequence into a string, specifying the separator to use between them.

Here's the output from a sample run. The length of the sequence and the numbers in it will be random each time you run it.

```
Stats for these 61 numbers:
The first 5 numbers: 85, 30, 58, 70, 60
The last 5 numbers: 40, 83, 75, 26, 75
The first is 85 and the last is 75
The smallest is 2, and the biggest is 99
The sum is 3444
The average is 56.46
```

LINQ queries aren't run until you access their results

Do this!

When you include a LINQ query in your code, it uses **deferred evaluation** (sometimes called lazy evaluation). That means the LINQ query doesn't actually do any enumerating or looping until your code executes a statement that ***uses the results of the query***. That sounds a little weird, but it makes a lot more sense when you see it in action. **Create a new console app** and add this code to your *Program.cs*:

```
var listOfObjects = new List<PrintWhenGetting>();
for (int i = 1; i < 5; i++)
    listOfObjects.Add(new PrintWhenGetting(i));

Console.WriteLine("Set up the query");
var result =
    from o in listOfObjects
    select o.Number;

Console.WriteLine("Run the foreach");
foreach (var number in result)
    Console.WriteLine($"Writing #{number}");


class PrintWhenGetting
{
    private int number;
    public PrintWhenGetting(int number)
    {
        this.number = number;
    }
    public int Number
    {
        get
        {
            Console.WriteLine($"Getting #{number}");
            return number;
        }
    }
}
```

The Console.WriteLine in the getter isn't called until the foreach loop actually executes. That's what deferred execution looks like.

```
Set up the query
Run the foreach
Getting #1
Writing #1
Getting #2
Writing #2
Getting #3
Writing #3
Getting #4
Writing #4
```

The query gets each value just before it writes the value.

The PrintWhenGetting class keeps track of a number and writes it to the console any time its getter is called.

Now run your app. Notice how the Console.WriteLine that prints "Set up the query" runs ***before*** the get accessor ever executes. That's because the LINQ query won't get executed until the `foreach` loop.

When you call ToList or another LINQ method that needs to access every element in the sequence, you'll get immediate evaluation.

If you need the query to execute right now, you can force **immediate execution** by calling a LINQ method that needs to enumerate the entire list—for example, the ToList method, which turns it into a List<T>. Add this line, and change the `foreach` to use the new List:

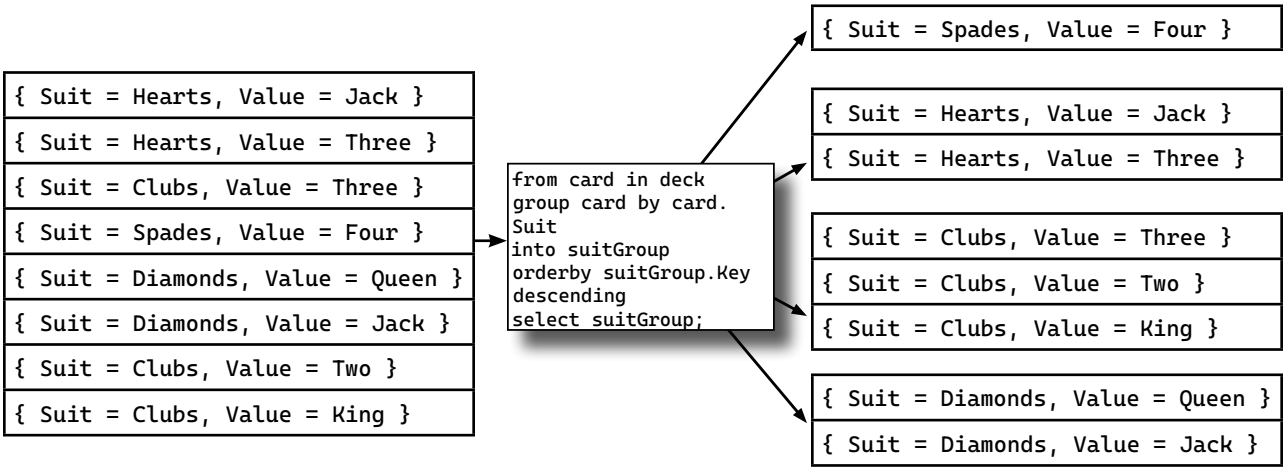
```
var immediate = result.ToList();

Console.WriteLine("Run the foreach");
foreach (var number in immediate)
    Console.WriteLine($"Writing #{number}");
```

```
Set up the query
Getting #1
Getting #2
Getting #3
Getting #4
Run the foreach
Writing #1
Writing #2
Writing #3
Writing #4
```

Run the app again. This time you'll see the get accessors called before the `foreach` loop starts executing—which makes sense, because ToList needs to access every element in the sequence to convert it to a List. Methods like Sum, Min, and Max also need to access every element in the sequence, so when you use them you'll force LINQ to do immediate execution as well.

Use a group query to separate your sequence into groups

Sometimes you really want to slice and dice your data. For example, Jimmy might want to group his comics by the decade they were published. Or maybe he wants to separate them by price (cheap ones in one collection, expensive ones in another). There are lots of reasons you might want to group your data together. That's where the **LINQ group query** comes in handy.



1 **Create a new console app and add the card classes and enums.**

Create a **new console app named CardLinq**. You'll be using cards, so you'll need the Deck, Card, and CardComparerByValue classes and the Suits and Values enums that you used in the MauiCards and TwoDecks projects at the end of Chapter 8.

Visual Studio: Go to the Solution Explorer panel, right-click on the project name and choose Add >> Existing Item, navigate to the folder where you saved the Two Decks project from Chapter 8, and add the files with the **Suit and Value enums**, then add the **Deck, Card, and CardComparerByValue classes**.

VSCode: navigate to the MauiCards project folder in Explorer or Finder and drag the files into your project.

Open any of the card files, click on the MauiCards namespace, and use F2 to **rename it to CardLinq**.

2 **Make your Card class sortable with IComparable<T>.**

We'll be using a LINQ `orderby` clause to sort groups, so we need the Card class to be sortable. Luckily, this works exactly like the List.Sort method. We learned in Chapter 8 that method works on objects that implement IComparable, so modify your Card class to **extend the IComparable interface**:

```
class Card(Values value, Suits suit) : IComparable<Card>
{
    public int CompareTo(Card? other)
    {
        return new CardComparerByValue().Compare(this, other);
    }

    // the rest of the class stays the same
```

We'll be using the LINQ Min and Max methods to find the highest and lowest card in each group, and they also use the IComparable interface.

3 **Modify the Deck.Shuffle method to support method chaining.**

The Shuffle class shuffles the deck. All you need to do to make it support method chaining is modify it to return a reference to the Deck instance that just got shuffled:

```
public Deck Shuffle()
{
    // The rest of the class stays the same

    return this;
}
```

When you make the Shuffle method return a reference to the same Deck object being shuffled, you can call it and then chain additional method calls to the result..

4 **Use a LINQ query with `group…by` clause to group the cards by suit.**

The top-level statements get 16 random cards by shuffling the deck, then use the LINQ Take method to pull the first 16 cards. Then they use a LINQ query with a **`group…by` clause** to separate the deck into smaller sequences, with one sequence for each suit in the 16 cards:

```
var deck = new Deck()
    .Shuffle()
    .Take(16);

var grouped =
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;

foreach (var group in grouped)
{
    Console.WriteLine(@$"Group: {group.Key}
    Count: {group.Count()}
    Minimum: {group.Min()}
    Maximum: {group.Max()}");
}
```

Now that the Shuffle method supports method chaining, you can chain the LINQ Take method right after it.

The Count, Min, and Max methods get information about each group.

Each group has a Key property that returns its key—in this case, a suit.

A `group…by` clause in a LINQ query separates a sequence into groups:

```
group card by card.Suit
    into suitGroup
```

The `group` keyword tells it which sequence contains the elements to group, the `by` keyword specifies the criteria used to determine the groups, and the `into` keyword declares a new variable that the other clauses can use to refer to the groups. The output of a group query is a sequence of sequences. Each group is a sequence that implements the IGrouping interface: IGrouping<Suits, Card> is a group of cards that uses a suit as its group key.

IDE Tip: Rename Anything!

5

Instead of adding a `using` directive to your top-level statements, **rename the MauiCards namespace to CardLinq**.

When you need to change the name of a variable, field, property, namespace, or class, you can use a really handy **refactoring tool** built into Visual Studio. Just right-click on it and choose Rename from the menu. When the IDE highlights it, edit its name—and the IDE will ***automatically rename it everywhere in the code***.

Use Rename to change the name of a **variable, field, property, class, or namespace** (and a few other things too!). When you rename one occurrence, the IDE changes its name everywhere it occurs in the code.

Anatomy of a Group Query

Let's take a closer look at how that group query works.

```
var grouped =
    from card in deck
```

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }

This random sample happens to start with two cards with the suit Hearts, followed by a card with the suit Clubs, then a Spades card, then two Diamonds cards.

The from clause works just like in the other LINQ queries you've used. It assigns the range variable "card" to each card in the sequence—in this case, the Deck that you shuffled and then pulled some cards from.

```
group card by card.Suit into suitGroup
```

The group...by clause splits the cards into groups. It includes "by card.Suit"—that specifies that the key for each group is the card's suit. It declares a new variable called suitGroup that the remaining clauses can use to work with the groups.

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

The group...by clause enumerates the sequence, creating new groups as it gets to each new key. So the groups are in the same order as the first occurrences of the suits that were in the random sample.

```
orderby suitGroup.Key descending
select suitGroup;
```

{ Suit = Spades, Value = Four }
{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

The group...by clause grouped the cards by card.Suit, so each group's key is the suit. That means all of the cards in each group have the same suit, and all of the cards with that suit are in the group. The orderby clause sorted the groups by key, which put them in the order they appear in the Suits enum (in reverse): Spades, Hearts, Clubs, and Diamonds.

The `group...by` clause creates a sequence of groups that implement the IGrouping interface. IGrouping extends IEnumerable and adds exactly one member: a property called Key. So each group is a sequence of other sequences—in this case, it's a group of Card sequences, where the key is the suit of the card (from the `Suits` enum). The full type of each group is IGrouping<Suits, Card>, which means it's a sequence of Card sequences, each of which has a Suits value as its key.

Use join queries to merge data from two sequences

Every good collector knows that critical reviews can have a big impact on values. Jimmy's been keeping track of reviewer scores from two big comic review aggregators, MuddyCritic and Rotten Tornadoes. Now he needs to match them up to his collection. LINQ to the rescue! The **`join`** keyword lets you **combine data from two sequences** using a single query. It compares items in one sequence with their matching items in a second sequence. The final result combines every pair that matches. Here's a join query that matches comics to a collection of Review objects:

```
from comic in Comic.Catalog
join review in reviews
on comic.Issue equals review.Issue
```

The join query compares the comic issue with the review issue and pairs up matching results. LINQ is smart enough to do that efficiently—it doesn't actually compare every pair of items unless it has to.

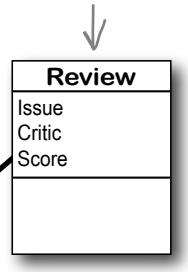
Let's take a closer look at how this join query works:

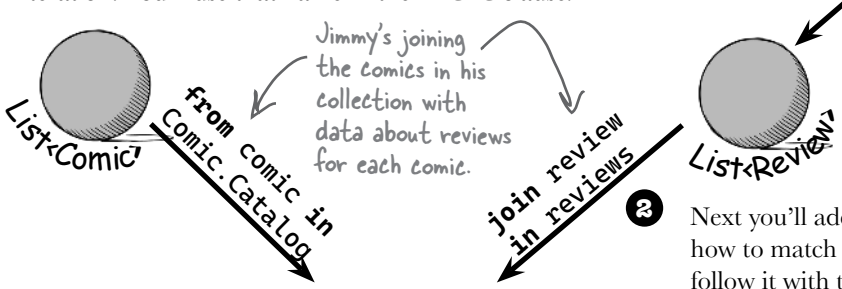
1 You'll start your query with the usual **`from`** clause. But instead of following it up with the criteria to determine what goes into the results, you'll add:

```
join name in collection
```

The **`join`** clause tells LINQ to enumerate both sequences to match up pairs with one member from each. It assigns **`name`** to the member it'll pull out of the joined collection in each iteration. You'll use that name in the **`where`** clause.

Jimmy's got his data in a collection of Review objects called "reviews"..\





2 Next you'll add the **`on`** clause, which tells LINQ how to match the two collections together. You'll follow it with the name of the member of the first collection you're matching, followed by **`equals`** and the name of the member of the second collection to match it up with.

```
on comic.Issue equals review.Issue
```

3 You'll continue the LINQ query with **`where`** and **`orderby`** clauses as usual. You could finish it with a normal **`select`** clause, but you usually want to return results that pull some data from one collection and other data from the other.

```
{ Name = "Woman's Work", Issue = 36, Critic =
  MuddyCritic, Score = 37.6 }
{ Name = "Black Monday", Issue = 74, Critic =
  RottenTornadoes, Score = 22.8 }
{ Name = "Black Monday", Issue = 74, Critic =
  MuddyCritic, Score = 84.2 }
{ Name = "The Death of the Object", Issue = 97,
  Critic = MuddyCritic, Score = 98.1 }
```

The result is a sequence of objects that have Name and Issue properties from Comic, but Critic and Score properties from Review. The result can't be a sequence of Comic objects, but it also can't be a sequence of Review objects, because neither class has all of those properties. The result is a *different* kind of type: ***an anonymous type***.

Use the new keyword to create anonymous types

You've been using the `new` keyword since Chapter 3 to create instances of objects. Every time you use it, you include a type (so the statement `new Guy();` creates an instance of the type Guy). You can also use the `new` keyword without a type to create an **anonymous type**. That's a perfectly valid type that has read-only properties, but doesn't have a name. The type we returned from the query that joined Jimmy's comics to reviews is an anonymous type. You can add properties to your anonymous type by using an object initializer. Here's what that looks like:

a-non-y-mous, adjective. not identified by name. *Secret Agent Dash Martin used his alias to become* ***anonymous*** *and keep the enemy spies from recognizing him.*

```
var whatAmI = new { Color = "Blue", Flavor = "Tasty", Height = 37 };
Console.WriteLine(whatAmI);
```

Try pasting that into a new console app and running it. You'll see this output:

```
{ Color = Blue, Flavor = Tasty, Height = 37 }
```

The LINQ query we just saw that joins Jimmy's comics with reviews returns an anonymous type. You'll add that query to an app later in the chapter.

Now hover over `whatAmI` in the IDE and have a look at the IntelliSense window:

```
whatAmI

(local variable) 'a? whatAmI

Types:
    'a is new { string Color, string Flavor, int Height }

'whatAmI' is not null here.
```

The IDE knows exactly what the type is: it's an object type with two string properties and an int property. It just doesn't have a name for the type. That's why it's an anonymous type.

The `whatAmI` variable is a reference type, just like any other reference. It points to an object on the heap, and you can use it to access that object's members—in this case, two of its properties:

```
Console.WriteLine($"My color is {whatAmI.Color} and I'm {whatAmI.Flavor}");
```

Besides the fact that they don't have names, anonymous types are just like any other types.

I bet that's why we're learning about the ***var*** *keyword now. We need it to declare anonymous types.*

Right! You use `var` to declare anonymous types.

The `var` keyword is really useful, and a lot of developers get in the habit of using it every time they declare a variable. But when you're using anonymous types, **you have to use it**—that's the ***only*** way to declare a variable that holds an anonymous type.

Sharpen your pencil

Joe, Bob, and Alice are some of the top competitive Go Fish players in the world. Here's an app with LINQ code that joins two arrays with anonymous types to generate a list of their winnings. Read through the code, figure out what the LINQ query does, and write the output that it writes to the console.

```
var players = new[]
{
    new { Name = "Joe", YearsPlayed = 7, GlobalRank = 21 },
    new { Name = "Bob", YearsPlayed = 5, GlobalRank = 13 },
    new { Name = "Alice", YearsPlayed = 11, GlobalRank = 17 },
};

var playerWins = new[]
{
    new { Name = "Joe", Round = 1, Winnings = 1.5M },
    new { Name = "Alice", Round = 2, Winnings = 2M },
    new { Name = "Bob", Round = 3, Winnings = .75M },
    new { Name = "Alice", Round = 4, Winnings = 1.3M },
    new { Name = "Alice", Round = 5, Winnings = .7M },
    new { Name = "Joe", Round = 6, Winnings = 1M },
};

var playerStats =
    from player in players
    join win in playerWins
    on player.Name equals win.Name
    orderby player.Name
    select new
    {
        Name = player.Name,
        YearsPlayed = player.YearsPlayed,
        GlobalRank = player.GlobalRank,
        Round = win.Round,
        Winnings = win.Winnings,
    };

foreach (var stat in playerStats)
    Console.WriteLine(stat);
```

Look closely at how these arrays are defined. We're using `var` and `new[]` to create implicity typed arrays that *hold anonymous types*. That means players and playerWins have a specific type. If we used a collection expression, we'd have to make them object[] arrays.

This code writes six lines to the console. We started you out by filling in the first two lines. Notice how both of those lines have the same name ("Alice"). A join query will find every match between the key properties in both sequences. If there are multiple matches, the results will include one element for each match. If there's a key in one input sequence that doesn't have a match in the other, it won't be included in the results.

{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }

{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }

there are no Dumb Questions

Q: Can you rewind a minute and explain what `var` is again?

A: Yes, definitely. The `var` keyword solves a tricky problem that LINQ brings with it. Normally, when you call a method or execute a statement, it's absolutely clear what types you're working with. If you've got a method that returns a string, for instance, then you can only store its results in a string variable or field.

But LINQ isn't quite so simple. When you build a LINQ statement, it might return an anonymous type that *isn't defined anywhere in your program*. Yes, you know that it's going to be a sequence of some sort. But what kind of sequence will it be? You don't know—because the objects that are contained in the sequence depend entirely on what you put in your LINQ query. Take this query, for example, from the code we wrote earlier for Jimmy. We originally wrote this:

```
IEnumerable<Comic> mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby -Comic.Prices[comic.Issue]
    select comic;
```

But then we changed the first line to use the `var` keyword:

```
var mostExpensive =
```

And that's useful. For example, if we changed the last line to this:

```
    select new {
       Name = comic.Name,
       IssueNumber = $"#{comic.Issue}"
    };
```

the updated query would return a different (but perfectly valid!) type—an anonymous type with two members, a string called Name and a string called IssueNumber. But we don't have a class definition for that type anywhere in our program! You don't actually need to run the program to see exactly how that type is defined, but the `mostExpensive` variable still needs to be declared with some type.

That's why C# gives us the `var` keyword, which tells the compiler, "OK, we know that this is a valid type, but we can't exactly tell you what it is right now. So why don't you just figure that out yourself and not bother us with it? Thanks so much."

Sharpen your pencil Solution

Joe, Bob, and Alice are some of the top competitive Go Fish players in the world. This LINQ code joins two arrays with anonymous types to generate a list of their winnings. Read through the code and write the output that it writes to the console.

```
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 5, Winnings = 0.7 }
{ Name = Bob, YearsPlayed = 5, GlobalRank = 13, Round = 3, Winnings = 0.75 }
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 1, Winnings = 1.5 }
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 6, Winnings = 1 }
```

IDE Tip: Refactoring Tools

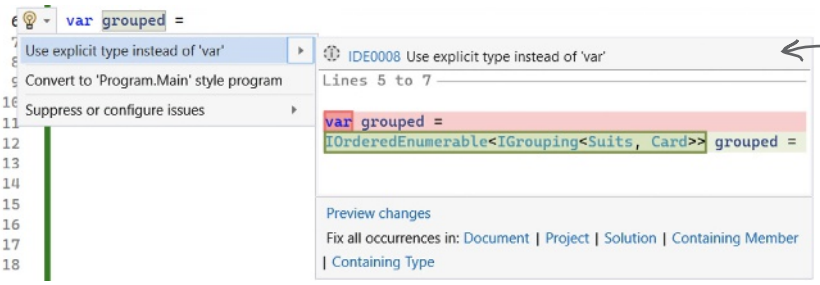
Switch variables between implicit and explicit types

When you're working with group queries, you'll often use the `var` keyword—not just because it's convenient, but because the type returned by the group query can be a little cumbersome.

```
var grouped =
   from card in deck
   group card by card.Suit into suitGroup
   orderby suitGroup.Key descending
   select suitGroup;
```

(local variable) IOrderedEnumerable<IGrouping<Suits, Card>>? grouped

But sometimes our code is actually easier to understand if we use the **explicit type**. Luckily, the IDE makes it easy to switch between the implicit type (`var`) and the explicit type for any variable. Just open the Quick Actions menu and choose **"Use explicit type instead of 'var'"** to convert the var to its explicit type.



You can use Quick Actions to change the implicit type "var" to an explicit type—in this case, IOrderedEnumerable <IGrouping<Suits, Card>>, which is definitely not a lot of fun to type!

You can also choose **"Use implicit type"** from the Quick Actions menu to change the variable back to `var`.

Extract methods

Your code will often be easier to read if you take a large method and break it into smaller ones. That's why one of the most common ways that developers refactor code is by ***extracting methods***, or taking a block of code from a large method and moving it into its own method. The IDE gives you a really useful refactoring tool to make that easy.

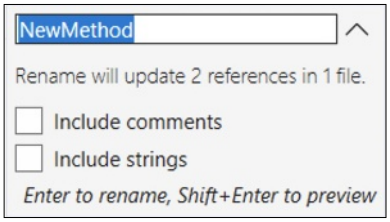
```
1 reference
void NewCardGame(IEnumerable<Player> players)
{
    var deck = new Deck().Shuffle();

    var grouped =
       from card in deck
       group card by card.Suit into suitGroup
       orderby suitGroup.Key descending
       select suitGroup;

    foreach (Player p in players)
```

Start by selecting a block of code. Then **choose Refactor >> Extract Method** from the Edit menu (Visual Studio) or **Extract Method** from the Quick Actions (VSCode).

As soon as you do, the IDE moves the selected code into a new method called NewMethod with a return type that matches the type of the code that's returned. In Visual Studio, it immediately jumps into the Rename feature so you can start typing a new method name.

In this screenshot, we selected the entire LINQ query from the card grouping project earlier in the chapter (but we put it inside a method because *the IDE **won't** extract a method from top-level statements*). After we extracted the method, this is what it looked like:

```
IOrderedEnumerable<IGrouping<Suits, Card>> grouped = NewMethod(deck);
```



Notice that it left a new explicitly typed **grouped** variable—the IDE figured out that the variable is used later in the code and left a variable in place. That's yet another example of how the IDE helps write cleaner code.

there are no Dumb Questions

Q: Can you give me a little more detail on how `join` works?

A: `join` works with any two sequences. Let's say you're printing T-shirts for football players, using a collection called `players` with objects that have a Name property and a Number property. What if you need a different design for players with double-digit numbers? You could pull out the players with numbers greater than 10:

```
var doubleDigitPlayers =
   from player in players
   where player.Number > 10
   select player;
```

Now what if you need to get their shirt sizes? If you have a sequence called `jerseys` whose items have a Number property and a Size property, a join would work really well for combining the data:

```
var doubleDigitShirtSizes =
   from player in players
   where player.Number > 10
   join shirt in jerseys
   on player.Number equals shirt.Number
   select shirt;
```

Q: That query will just give me a bunch of objects. What if I want to connect each player to their shirt size, and I don't care about the number at all?

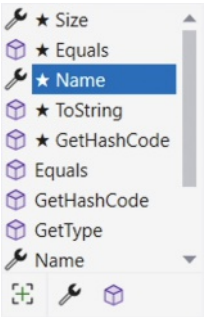
A: That's what anonymous types are for—you can construct an anonymous type that only has the data you want in it. It lets you pick and choose from the various collections that you're joining together too.

So you can select the player's name and the shirt's size, and nothing else:

```
var doubleDigitShirtSizes =
    from player in players
    where player.Number > 10
    join shirt in jerseys
    on player.Number equals shirt.Number
    select new {
        player.Name,
        shirt.Size
    };
```

The IDE is smart enough to figure out exactly what results you'll be creating with your query. If you create a loop to enumerate through the results, as soon as you type the variable name, the IDE will pop up an IntelliSense list:

```
foreach (var size in doubleDigitShirtSizes)
    size.
```



The list has Name and Size in it. If you added more items to the `select` clause, they'd show up in the list, too, because the query would create a different anonymous type with different members.

Q: How do I write a method that returns an anonymous type?

A: You don't. Methods cannot return anonymous types. C# doesn't give you a way to do that. You can't declare a field or a property with an anonymous type either. You also can't use an anonymous type for a parameter in a method or a constructor—that's why you can't use the `var` keyword with any of those things.

And when you think about it, these things make sense. Whenever you use `var` in a variable declaration, you always have to include a value, which the C# compiler or IDE uses to figure out the type of the variable. If you're declaring a field or a method parameter, there's no way to specify that value—which means there's no way for C# to figure out the type. (Yes, you can specify a value for a property, but that's not really the same thing—technically, the value is set just before the constructor is called.)

You can only use the var keyword when you're declaring a variable. You can't use var with a field or a property, or write a method that returns an anonymous type or takes one as a parameter.

Exercise

Use what you've learned about LINQ so far to **build a *new console app* called** JimmyLinq that organizes Jimmy's comic collection and keeps track of critics' reviews. Start by **adding the Comic class**. Then **add a** Critics **enum** with two members, MuddyCritic and RottenTornadoes, and **a `PriceRange` enum** with two members, Cheap and Expensive. Finally, **add a Review class** with three properties: int Issue, Critics Critic, and double Score.

You'll need data, so add a new static field to the Comic class that returns a sequence of reviews:

```
public static readonly IEnumerable<Review> Reviews = [
   new Review(36, Critics.MuddyCritic, 37.6),
   new Review(74, Critics.RottenTornadoes, 22.8),
   new Review(74, Critics.MuddyCritic, 84.2),
   new Review(83, Critics.RottenTornadoes, 89.4),
   new Review(97, Critics.MuddyCritic, 98.1),
];
```

The Review class has a primary constructor with arguments for issue, critic, and score that are returned by the properties.

Here are the top-level statements and methods they call:

Look closely at this while loop. It uses a switch to determine which method to call. The methods return true, setting "done" to true and the while loop to do another iteration. If the user presses any other key, it sets "done" to false and ends the loop.

```
using JimmyLinq;

var done = false;
while (!done) {
    Console.WriteLine(
        "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
    switch (Console.ReadKey(true).KeyChar.ToString().ToUpper()) {
        case "G":
        done = GroupComicsByPrice();
            break;
        case "R":
            done = GetReviews();
            break;
        default:
            done = true;
            break;
    }
}

bool GroupComicsByPrice() {
    var groups = ComicAnalyzer.GroupComicsByPrice(Comic.Catalog, Comic.Prices);
    foreach (var group in groups) {
        Console.WriteLine($"{group.Key} comics:");
        foreach (var comic in group)
            Console.WriteLine($"#{comic.Issue} {comic.Name}: {Comic.Prices[comic.Issue]:c}");
        }
    return false;
}

bool GetReviews() {
    var reviews = ComicAnalyzer.GetReviews(Comic.Catalog, Comic.Reviews);
    foreach (var review in reviews)
         Console.WriteLine(review);
    return false;
}
```

The `foreach` loops in the GroupComicsByPrice method are *nested*: one loop is inside the other. The outer loop prints information about each group, and the inner one enumerates the group.

The GroupComicsByPrice and GetReviews methods call methods in the static ComicAnalyzer class (which you'll write) that run LINQ queries.

Make sure the new classes and enums are in the JimmyLinq namespace.

Your job is to **create a static class called ComicAnalyzer** with three static methods (two of which are public):

- A private static method called CalculatePriceRange takes a Comic reference and returns PriceRange.Cheap if the price is under 100 and PriceRange.Expensive otherwise.
- GroupComicsByPrice orders the comics by price, then groups them by CalculatePriceRange(comic) and returns a sequence of groups of comics (IEnumerable<IGrouping<PriceRange, Comic>>).
- GetReviews orders the comics by issue number, then does the join you saw earlier in the chapter and returns a sequence of strings like this: `MuddyCritic rated #74 'Black Monday' 84.20`

Exercise Solution

Make sure all of the classes and enums that you add are in the *JimmyLinq namespace* by including `namespace JimmyLinq;` at the top of each file.

First add the Critics and PriceRange enums:

```
enum Critics
{
    MuddyCritic,
    RottenTornadoes,
}

enum PriceRange
{
    Cheap,
    Expensive,
}
```

Then add the Review class:

```
class Review(int issue, Critics critic, double score)
{
    public int Issue { get { return issue; } }
    public Critics Critic { get { return critic; } }
    public double Score { get { return score; } }
}
```

Once you have those, you can add the static ComicAnalyzer class with a private PriceRange method and public GroupComicsByPrice and GetReviews methods:

We've *intentionally* included a bug in this CalculatePriceRange method, so make sure the code in your method matches this solution.

```
static class ComicAnalyzer
{
   private static PriceRange CalculatePriceRange(Comic comic)
   {
      if (Comic.Prices[comic.Issue] < 100)
         return PriceRange.Cheap;
      else
         return PriceRange.Expensive;
   }

   public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
                        IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
   {
      IEnumerable<IGrouping<PriceRange, Comic>> grouped =
         from comic in comics
         orderby prices[comic.Issue]
         group comic by CalculatePriceRange(comic) into priceGroup
         select priceGroup;

      return grouped;
   }

   public static IEnumerable<string> GetReviews(
                                    IEnumerable<Comic> comics, IEnumerable<Review> reviews)
   {
      var joined =
         from comic in comics
         orderby comic.Issue
         join review in reviews on comic.Issue equals review.Issue
         select $"{review.Critic} rated #{comic.Issue} '{comic.Name}' {review.Score:0.00}";

      return joined;
   }
}
```

Can you spot the bug? It's subtle...

The refactoring tools that we showed you earlier in the chapter make it easier to get the return type of the GroupComicsByPrice method right.

We asked you to order the comics by price, then group them. That causes each group to be sorted by price, because the groups are created in order as the group...by clause enumerates the sequence.

This is really similar to the join query that we explained earlier in the chapter.

Did you run into a compiler error about "inconsistent accessibility" telling you a property or return type is less accessible than the method? That happens when a class is marked public but has members that are marked internal (or not marked at all, because internal is the default if you leave off the access modifier). So make sure that *none of the classes or enums are marked public*.

Bullet Points

- The **`group`**…**`by`** clause tells LINQ to group the results. When you use it, LINQ creates a sequence of group sequences of type IGrouping<TKey, TValue>.
- Every group contains members that have one member in common, called the group's **key**. Use the **`by`** keyword to specify the key for the group. Each group sequence has a Key member that contains the group's key.
- Join queries use an **`on`**…**`equals`** clause to tell LINQ how to match the pairs of items.
- Use a **`join`** clause to tell LINQ to combine two collections into a single query. When you do this, LINQ compares every member of the first collection with every member of the second collection, including the matching pairs in the results.
- When you're doing a join query, you usually want a set of results that includes some members from the first collection and other members from the second collection. The **`select`** clause lets you build custom results from both of them.
- Use **`select new`** to construct custom LINQ query results with an anonymous type that includes only the specific properties you want in your result sequence.
- LINQ queries use **deferred evaluation** (sometimes called lazy evaluation), which means they don't run until a statement uses the results of the query.
- Use the **`new`** keyword to create an instance of an **anonymous type**, or an object with a well-defined type that doesn't have a name. The members specified in the **`new`** statement become automatic properties of the anonymous type.
- Use the **Rename feature** in the IDE to easily rename every instance of a variable, field, property, class, or namespace at once.
- Use the IDE's Quick Actions menu to change a **`var`** declaration to an **explicit type**, or back to a **`var`** (or an implicit type) again.
- One of the most common ways that developers refactor code is by **extracting methods**. The IDE's Extract Method feature makes it very easy to move a block of code into its own method.
- You can **only use the `var` keyword** when you're declaring a variable. You can't write a method that returns an anonymous type, or which takes one as a parameter, or use one with a field or a property.

Back in Chapter 3, you learned about prototypes, or early versions of games that you can play, test, learn from, and improve—and you saw how that idea can work with any kind of project, not just games. The same thing applies to testing. Sometimes the idea of testing software can seem a little abstract. Thinking about how game developers test their games can help us get used to the idea, and can make the concept of testing feel more intuitive.

Game Design...and Beyond

Testing

As soon as you've got a playable prototype of your game, you're ready to think about video game ***testing***. Getting people to try out your game and give you feedback on it can make all the difference between a game that everybody loves playing and one that frustrates new users and gives an unsatisfying experience. If you've ever played a game that made you feel lost about what you were supposed to be doing, or one where the puzzles seemed unsolvable, then you know what happens when a game doesn't get enough playtesting.

Think about these common ways to do testing as you design and make games (or any apps!).

Individual playtesting: Ask people you know to play the game, preferably with you watching. The most informal way to do playtesting is to just ask a friend to play the game and talk about their experience as they're playing. It sometimes helps if you don't give them very much instruction and then pay special attention if your playtesters get stuck. That will help you to understand if the point of your game is well understood and will make it easier for you to see if some of the game mechanics you've put in place are not obvious enough to a user. Write down all of the feedback testers give you so you can fix any design problems that they identify.

Getting individual feedback can be done informally or in a more formal setting where you've drawn up a list of tasks you want the user to perform and a questionnaire to capture feedback about the game. It should be done frequently as you're adding new features, and you should ask people to playtest your game as early in development as possible to make sure you find design problems when they are easiest to fix.

Beta programs: When you are ready to have a larger audience check out the game, you can ask a larger group of people to play it before you open it up to the general public. Beta programs are great for finding problems with load and performance in your game. Usually, feedback from beta programs is analyzed through game logs: you'll have playtesters sign up for a beta test so you can ask them about their experience afterward and get their help troubleshooting issues you find in the log.

Structured quality assurance (QA) tests: Most games have dedicated tests that are run as part of development. These tests are based on an understanding of how the game is supposed to work, and they can be automated or manual. The goal of this kind of testing is to make sure that the product works as intended. When a quality assurance test is run, the idea is to find as many bugs as possible before a user has to deal with them.

Most development teams try to automate as much of their testing as possible, and run those tests before each commit. That way, they know if they inadvertently introduced a bug when they were making a fix or adding a new feature.

Unit tests help you make sure your code works

We intentionally left a bug in the code we gave you…but is that the *only* bug in the app? It's easy to write code that doesn't do exactly what you intended it to do. Luckily, there's a way for us to find bugs so we can fix them. **Unit tests** are *automated* tests that help you make sure your code does what it's supposed to do. Each unit test is a method that makes sure that a specific part of the code (the "unit" being tested) works. If the method runs without throwing an exception, it passes. If it throws an exception, it fails.

Visual Studio and VSCode have built-in unit testing tools to help you write your tests and track which ones pass or fail. The unit tests in this book will use **MSTest**, a unit test framework—a set of classes that give you the tools to write unit tests—developed by Microsoft.

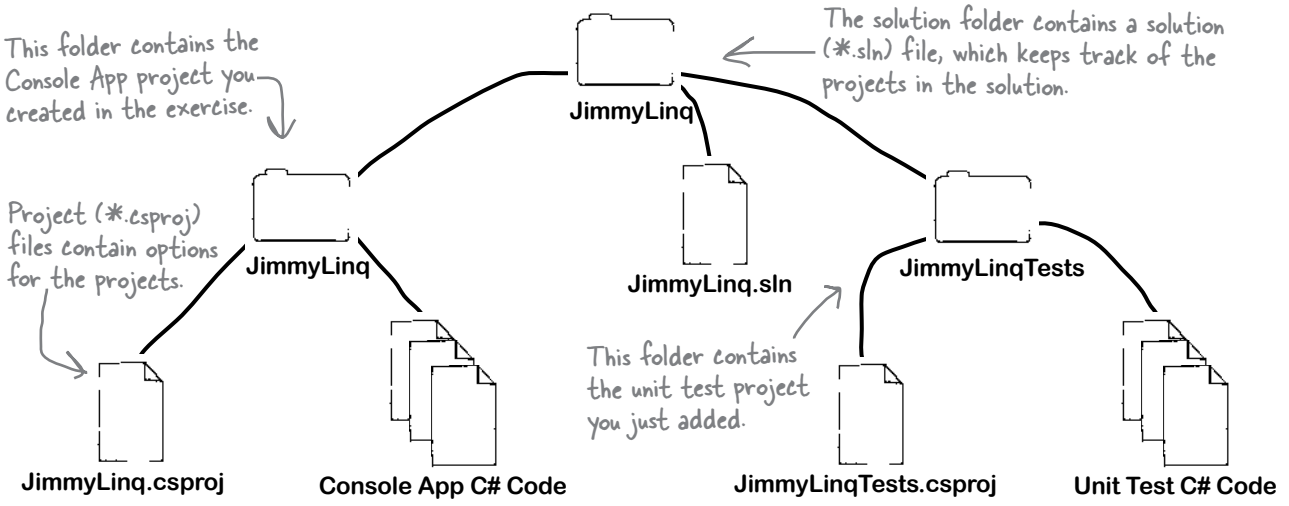
Add a second project to your solution for the unit tests

Back in Chapter 2, we talked about the folders your C# project uses: a folder for the solution and the *.sln* file, and another folder underneath it with the same name that contains the *.csproj* file and C# code files.

Most large programs have a **suite** of tests, or a series of **test classes** that contain the individual unit tests that make sure most or all of the code in the app works. Each test class contains a set of individual **test methods** that each check one specific behavior in the app—like making sure that a public method in a class returns the correct value when it's passed specific values. Let's add a new MSTest Test Project to your solution:

1. Go to the Solution Explorer and **right-click on the solution name** in the top row (not the project name in the second row).
2. In ***Visual Studio***, choose **Add >> New Project**. In ***VSCode***, choose **New Project** (or click the + button at the right side of the row with the solution name).
3. Search for the **MSTest Test Project template** and select it.
4. Name your new test project **JimmyLinqTests**. In VSCode, choose the default directory.

The IDE will create a new project folder underneath your solution folder. The source code folders and files for your app should now look like this:



Start writing your first test method

Expand the JimmyLinqTests project in the Solution Explorer. It has one C# code file, *UnitTest1.cs*, which contains a class called UnitTest1 with a method called TestMethod1. The class and method have **attributes** above them. An attribute is a way to add extra information to a class or member. The `[TestClass]` attribute above a class means that it's a test class, and the `[TestMethod]` attribute above a method means it's a test method.

Click on the class name, press F2, and **rename the class ComicAnalyzerTests**. Click on the file in the Solution Explorer, press F2, and **rename the file *ComicAnalyzerTests.cs***. Change the name of the test method to ComicAnalyzerShouldGroupComics. Your class should now look like this:

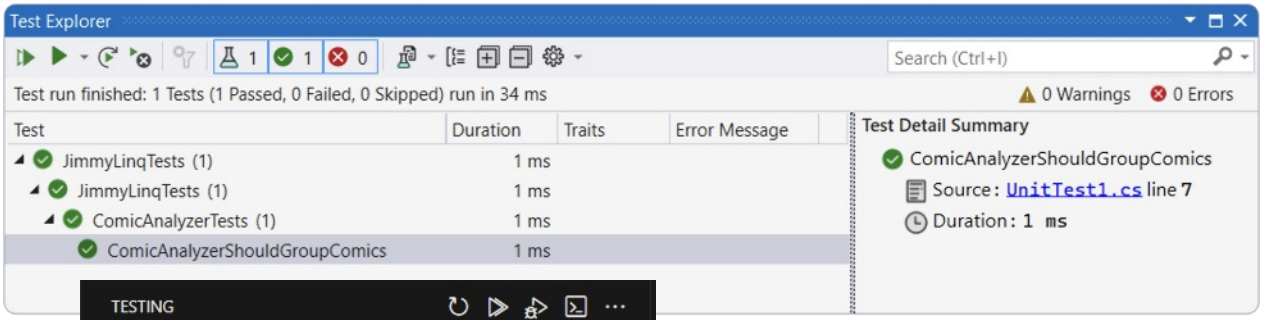
```
namespace JimmyLinqTests;

[TestClass]
public class ComicAnalyzerTests
{
    [TestMethod]
    public void ComicAnalyzerShouldGroupComics()
    {
    }
}
```

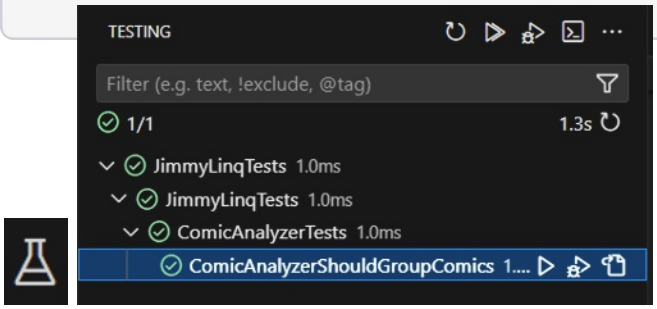
If VSCode tells you that no tests have been found in the workspace yet, *rebuild your solution* by opening the Command Palette (Ctrl+Shift+P) and choosing .NET: Rebuild.

Run the unit test

When you run your unit tests, each test runs just like any other C# code. You can run a single test, or run all of the tests. You can also debug them in the IDE. If a test method reaches the end without any problems, it **passes**. If it throws an exception, it **fails**. The ComicAnalyzerShouldGroupComics test method doesn't do anything, so it will pass.

Open the Test Explorer window. In Visual Studio, **choose Test Explorer (Ctrl+E T) from the Test menu**. In VSCode, **choose Testing from the View menu** or click the button on the left that looks like a beaker.





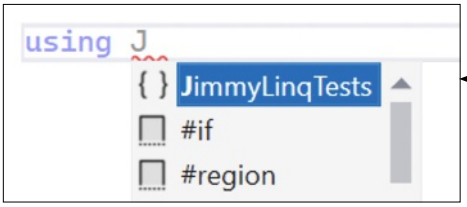
Click the **Run All Tests button** (or) to run all of the unit tests in your JimmyLinqTests project.

You can also **debug the tests** so you can set breakpoints. In Visual Studio, click the dropdown next to the run button ( ) and choose Debug. In VSCode, click the Debug Tests button ( ).When you expand the tests, you'll see buttons to run or debug each individual test.

Give your unit tests access to the classes they're testing

Your solution now has two projects: the JimmyLinq project has the app with classes that you want to test, and the JimmyLinqTests project will have the unit tests. The unit tests will be methods that use the classes in the JimmyLinq project. The tests will call those classes' public methods, properties, and other members to make sure they work.

There's just one problem. Open *UnitTest1.cs* in your unit test project and **try to add a `using` statement** to the top:



Try to add `using JimmyLinq;` to the top of the *UnitTest1.cs* file in the unit test project. Uh-oh—something went wrong! The JimmyLinq namespace doesn't show up in the IntelliSense window.

The classes in your JimmyLinqTests project can't access the ones in the JimmyLinq project yet.

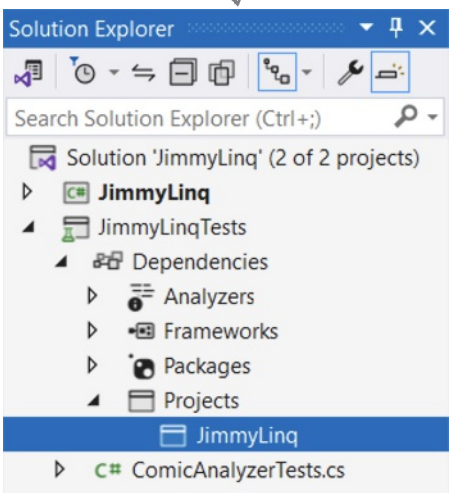
Add a project reference so the unit tests can access the classes they need to test

When you have two different projects in the same solution, they're *independent*: the classes in one project can't use classes in another project. If you want one project in a multiproject solution to access another project, you need to add a **project reference** to it. You need the unit test methods in the JimmyLinqTests project to use classes from JimmyLinq, so you'll ***modify the JimmyLinqTests project to add a reference to the JimmyLinq project***.

- **Visual Studio:** Expand the JimmyLinqTests project in the Solution Explorer, right-click on Dependencies, and choose **Add Project Reference** from the menu. Check the box for the JimmyLinq project.
- **Visual Studio Code:** Right-click on the project name in the Solution Explorer and choose **Add Project Reference**. VSCode displays the other projects in the solution. Click on the JimmyLinq project.

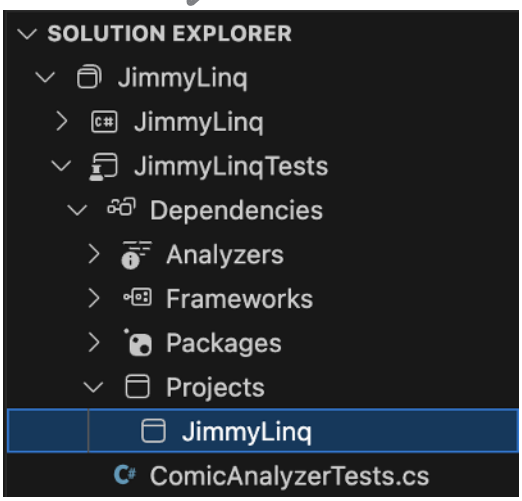
Here's what your Solution Explorer should look like after you add the project reference:

Visual Studio



Expand the Dependencies section under the JimmyLinqTests project in the Solution Explorer, then expand Projects. You should see a reference to the JimmyLinq project. Now you can use any public classes from the JimmyLinq project in your new unit test project.

Visual Studio Code



One project can only access public classes in another project

Try adding `using JimmyLinq;` to the top of the file with the test class again. Do you see JimmyLinq in the IntelliSense window yet? You might not.

You'll only be able to add the `using` directive to use classes from the JimmyLinq namespace if that namespace has at least one public class with at least one public member.

An internal class or member can only be used by other classes in a project. A public class or member can be used by classes in other projects as well.

If you're using Visual Studio, the JimmyLinq project probably has no public classes, enums, or other members because classes are created with the internal access modifer. If you're using VSCode, classes are created with the public access modifier. An internal class or member can only be used by other classes in a project. A public class or member can be used by classes in other projects as well.

Make your JimmyLinq classes and enums public

Your unit tests need to be able to access all of the classes in your JimmyLinq project. **Modify the Critics enum to make it public** by adding the public access modifier:

```
public enum Critics {
    MuddyCritic,
    RottenTornadoes,
}
```

If you add or remove unit tests or change their names, you might not see those changes in the Test Explorer until you *rebuild your unit test project*.

Then **rebuild your JimmyLinq solution**:

- Visual Studio: Choose Build >> Rebuild Solution from the menu (or just run your app).
- VSCode: Choose .NET: Rebuild from the Command Palette (or just run your app).

After your app is rebuilt, **add the `using JimmyLinq;` directive** to the top of the file with your test class. IntelliSense should now auto-complete JimmyLinq, and the `using` directive will work.

Now change the ComicAnalyzer declaration to make it public:

```
public static class ComicAnalyzer
```

Uh-oh—something's wrong. Did you get a bunch of "Inconsistent accessibility" compiler errors?

```
Inconsistent accessibility: return type 'IEnumerable<IGrouping<PriceRange, Comic>>' is less accessible than
method 'ComicAnalyzer.GroupComicsByPrice(IEnumerable<Comic>, IReadOnlyDictionary<int, decimal>)'
```

If you did, it's because ComicAnalyzer is public, but it exposes members that have no access modifiers, which makes them internal—so other projects in the solution can't see them. **Add the public access modifier** to ***every class and enum*** in the JimmyLinq project. Now your solution will build again.

Classes or class members marked internal can only be used by other classes in a project. To make something accessible by other projects, mark it public.

Use the Arrange-Act-Assert pattern to write an effective test

The whole purpose of a test method is to make sure that a method, property, or other public member of a class does what it's supposed to. Luckily, there's a really useful pattern to help you write great unit tests. A **pattern** is an approach that you can use over and over again when you're planning out how to write code. It's not ready-made code that you copy and paste—it's an ***idea or guideline*** to help guide you in writing your code.

There's a really useful pattern to help you write unit tests called **Arrange-Act-Assert** (or **AAA**). It has three parts:

- **Arrange:** Set up objects and other data to get ready for the part of the code that you're testing.
- **Act:** Do something with that data, like call a method or set a property.
- **Assert:** Make sure that the results are what you expect.

Here's a useful way to start coding a unit test—add comments that describe how its arrange, act, and assert will work:

```
[TestMethod]
public void ComicAnalyzerShouldGroupComics()
{
    // Arrange: Set up some comics and prices

    // Act: Call the method that groups comics by price

    // Assert: Check the groups and make sure they're correct
}
```

Starting your test by writing comments that explain how it will add, act, and assert is a great way to plan how it will work—and it makes the test a lot easier to write.

Use Assert.AreEqual to check that a calculated value matches an expected value

Unit tests use **assertions**, or methods that check ***actual results*** to see if they match ***expected results***. If the values don't match, the assertion throws an exception and the test fails. If no assertions fail, then the test passes. Here are some useful assertions—they're all **static methods in the Assert class** that's part of MSTest:

- Assert.AreEqual takes two arguments, an expected result (what you think the code should do) and an actual result (what it actually does), and throws an exception if they're not equal.
- Assert.AreSame is similar to Assert.AreEqual, except it checks if two references point to the same object.
- Assert.IsTrue and Assert.IsFalse take one argument and throw an exception if it's not true or false.
- Assert.Fail fails immediately. It's especially useful when paired with `if/else` or `switch` statements.

Most Assert methods have overloads that let you add a message that's displayed in the test results if the assertion fails. That can help you quickly figure out where your test failed, so you can start to sleuth out the problem.

When you're writing unit tests, it's always a good idea to take a few minutes to think through how they'll work before you start coding.

Brain Power

Think through how this unit test should work. What statements would you write to arrange your objects so you have data to test with? What will the act section do? How will the assert section check to make sure that everything worked as expected?

Finish your first unit test

Here's the code for your unit test method. First its **arrange** part sets up some **test data**: a sequence of comics and a Dictionary of prices. Then its **act** calls the GroupComicsByPrice using that test data to create groups. Finally, its **assert** uses the Assert.AreEqual method to check if values are correct.

```
namespace JimmyLinqTests;

using JimmyLinq;

[TestClass]
public class ComicAnalyzerTests
{
    [TestMethod]
    public void ComicAnalyzerShouldGroupComics()
    {
        // Arrange: Set up some comics and prices
        IEnumerable<Comic> testComics = [
             new Comic("Issue 1", 1),
             new Comic("Issue 2", 2),
             new Comic("Issue 3", 3),
        ];

        Dictionary<int, decimal> prices = new()
        {
            { 1, 20M },
            { 2, 10M },
            { 3, 1000M },
        };

        // Act: Call the method that groups comics by price
        var groups = ComicAnalyzer.GroupComicsByPrice(testComics, prices);

        // Assert: Check the groups and make sure they're correct
        Assert.AreEqual(2, groups.Count());
        Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
        Assert.AreEqual(2, groups.First().First().Issue);
        Assert.AreEqual("Issue 2", groups.First().First().Name);
    }
}
```

When you give your test methods descriptive names, it helps you figure out what went wrong if they fail.

The "Arrange" portion of this test sets up *test data*: a sequence of three comics and a Dictionary with three prices. It will pass this test data into the GroupComicsByPrice method.

Here's a useful C# shortcut! It's called a *target-typed new expression*. When a variable's type is already known—like when you specify it in the declaration—you can just use `new()` to instantiate a class. C# already knows it's a Dictionary<int, decimal>, so you can save yourself a few keystrokes.

These are the *expected results*. Our test data has three comics: two priced under 100 and one priced over 100. So it should create two groups, a group with two cheap comics followed by a group with one expensive comic.

The groups are sorted in ascending price order, so the first item in the first group should be issue #2.

Open the Test Explorer and **press the Run All Tests button**. Uh-oh—***your test failed!*** Look for this error:

```
Test method JimmyLinqTests.ComicAnalyzerTests.ComicAnalyzerShouldGroupComics threw exception:
System.Collections.Generic.KeyNotFoundException: The given key '2' was not present in the dictionary.
```

This is the result of a **failed unit test**. Look for the ⊗ 1 icon in Visual Studio at the top of the Test Explorer or ⊗ 0/1 in VSCode at the bottom of the Test Explorer—that's how you see a count of your failed unit tests.

Were you surprised that the unit test failed? Can you figure out what went wrong?

Sleuth it Out

Unit testing is all about discovering places where your code doesn't act the way you expect and sleuthing out exactly what went wrong. Sherlock Holmes once said, "It is a capital mistake to theorize before one has data." So let's get data!

We'll start with the assertions

A good place to start is always the assertions that you included in the test, because they tell you what specific code you're testing, and how you expect it to behave. Here are the assertions from your unit test:

```
Assert.AreEqual(2, groups.Count());
Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
Assert.AreEqual(2, groups.First().First().Issue);
Assert.AreEqual("Issue 2", groups.First().First().Name);
```

When you look at the test data that you're feeding into the GroupComicsByPrice method, these assertions look correct. It really should return two groups. The first one should have the key PriceRange.Cheap. The groups are sorted by ascending price, so the first comic in the first group should have Issue = 2 and Name = "Issue 2"—and that's exactly what those assertions are testing. So if there's a problem, it's not here—these assertions really do seem to be correct.

Now let's have a look at the stack trace

You've seen plenty of exceptions by now. Each exception comes with a **stack trace**, or a list of all of the method calls the program made right up to the line of code that threw it. If it's in a method, it shows what line of code called that method, and what line called that one, all the way up to the top-level statements. **Open the stack trace** for your failed unit test:

- Visual Studio: Open the Test Explorer (View >> Test Explorer), click on the test, and scroll in the Test Detail Summary.
- VSCode: Go to the **Test Results panel** on the right that lists all of the test runs. Expand the most recent test run at the top, then click on the test failure to see the stack trace.

The stack trace will start like this (on Mac you'll see fully qualified class names like JimmyLinq.ComicAnalyzer):

```
Dictionary`2.get_Item(TKey key)
ComicAnalyzer.CalculatePriceRange(Comic comic)line7
<>c.<GroupComicsByPrice>b__1_1(Comic comic)line19
```

Visual Studio has handy underlined links that jump right to the code. VSCode stack traces look a little different, but have the same information.

Stack traces look a little weird at first, but once you get used to them they've got a lot of useful information. Now we know that the test failed because an exception related to Dictionary keys was thrown *somewhere* inside CalculatePriceRange.

Use the debugger to gather clues

Add a **breakpoint to the first line** of the CalculatePriceRange method: `if (Comic.Prices[comic.Issue] < 100)`

Then **debug your unit tests**. Visual Studio: choose Test >> Debug All Test. VSCode: go to the Testing panel (View >> Testing), right-click JimmyLinqTest, and choose Debug Test. Hover over `comic.Issue`—its value is 2. But wait a minute! The Comic.Prices Dictionary **doesn't have an entry** with the key 2. ***No wonder it threw the exception!***

Now we know ***how to fix the problem***:

- Add a second parameter to the CalculatePriceRange method:
  `... CalculatePriceRange(Comic comic, IReadOnlyDictionary<int, decimal> prices)`
- Modify the first line to use the new parameter: `if (prices[comic.Issue] < 100)`
- Modify the LINQ query: `group comic by CalculatePriceRange(comic, prices) into priceGroup`

`Run` your test again. This time it passes! Visual Studio will show this: 1 0 VSCode will show this: 1/1

Write a unit test for the GetReviews method

The unit test for the GroupComicsByPrice method used MSTest's static Assert.AreEqual method to check expected values against actual ones. The GetReviews method *returns a sequence of strings*, not an individual value. We *could* use Assert.AreEqual to compare individual elements in that sequence, just like we did with the last two assertions, using LINQ methods like First to get specific elements…but that would take a LOT of code.

Luckily, MSTest has a better way to compare collections: the **CollectionAssert class** has static methods for comparing expected versus actual collection results. So if you have a collection with expected results and a collection with actual results, you can compare them like this:

```
CollectionAssert.AreEqual(expectedResults, actualResults);
```

If the expected and actual results don't match, the test will fail. Go ahead and **add this test** to validate the ComicAnalyzer.GetReviews method:

```
[TestMethod]
public void ComicAnalyzerShouldGenerateAListOfReviews()
{
    // Arrange: Set up test data with comics and reviews
    IEnumerable<Comic> testComics = [
        new("Issue 1", 1),
        new("Issue 2", 2),
        new("Issue 3", 3),
    ];

    IEnumerable<Review> testReviews = [
        new(1, Critics.MuddyCritic,14.5),
        new(1, Critics.RottenTornadoes,59.93),
        new(2, Critics.MuddyCritic,40.3),
        new(2, Critics.RottenTornadoes,95.11),
    ];

    // Act: Call the method that gets the reviews
    var actualResults =
             ComicAnalyzer
               .GetReviews(testComics, testReviews)
               .ToList();

    // Assert: Check the results to make sure it got the correct reivews
    var expectedResults = new[]
    {
        "MuddyCritic rated #1 'Issue 1' 14.50",
        "RottenTornadoes rated #1 'Issue 1' 59.93",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "RottenTornadoes rated #2 'Issue 2' 95.11",
    };
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Here are more *target-typed expressions*. We told the C# compiler the collection expression types: testComics is an sequence of Comic references, and testReviews is a sequence of Review references.

So instead of writing this:

`new Comic("Issue 2", 2)`

we *leave out the type* and write this instead:

`new("Issue 2", 2)`

The compiler already knows that it's creating a Comic object, so we can leave off the class name. Many developers use this shortcut because they think it makes the code less cluttered and easier to read. Do you agree?

GetReviews returns a sequence of strings. This string array contains the results we expect it to return, so we can use CollectionAssert to compare the expected results with the actual results.

We could have used a collection initializer to create the expected results. Do you think that would have been easier to read?

Now run your tests again. You should see two unit tests pass.

Write unit tests to handle edge cases and weird data

Back in Chapter 4, we learned about **edge cases**, or situations or values that are not typical and may cause weird behavior. In the real world, data is messy. For example, we've used review scores between 0 and 100. Did you assume those were the only values allowed? That's definitely the way some review websites in the real world operate. What if we get some weird review scores—like negative ones, or really big ones, or zero? What if we get more than one score from a reviewer for an issue? Even if these things aren't *supposed to* happen, they *might* happen.

We want our code to be **robust**, which means that it handles problems, failures, and especially bad input data well. So let's build a unit test that passes some weird data to GetReviews and makes sure it doesn't break:

```
[TestMethod]
public void ComicAnalyzerShouldHandleWeirdReviewScores()
{
    // Arrange: Set up test data with weird edge cases
    IEnumerable<Comic> testComics = [
        new("Issue 1", 1),
        new("Issue 2", 2),
        new("Issue 3", 3),
    ];

    IEnumerable<Review> testReviews = [
        new(1, Critics.MuddyCritic, -12.1212),
        new(1, Critics.RottenTornadoes, 391691234.48931),
        new(2, Critics.RottenTornadoes, 0),
        new(2, Critics.MuddyCritic, 40.3),
        new(2, Critics.MuddyCritic, 40.3),
        new(2, Critics.MuddyCritic, 40.3),
        new(2, Critics.MuddyCritic, 40.3),
    ];

    // Act: Call the method that gets the reviews
    var actualResults = ComicAnalyzer.GetReviews(testComics, testReviews).ToList();

    // Assert: Make sure it got the weird edge cases right
    List<string> expectedResults = [
        "MuddyCritic rated #1 'Issue 1' -12.12",
        "RottenTornadoes rated #1 'Issue 1' 391691234.49",
        "RottenTornadoes rated #2 'Issue 2' 0.00",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
    ];
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Can our code handle negative numbers? Really big numbers? Zero? These are great cases for a unit test to check.

What if we got exactly the same review from the same critic several times in a row? It seems obvious how the code should handle it, but that doesn't mean the code actually does.

Always take time to write unit tests for *edge cases and weird data*. Think of them as "must-have" and not "nice-to-have" tests. The point of unit testing is to cast the widest possible net to catch bugs, and these kinds of tests are really effective for that.

It's really important to add unit tests that handle edge cases and weird data. They can help you spot problems in your code that you wouldn't find otherwise.

ro-bust, adjective.
sturdy in construction. *The bridge's* ***robust*** *design allowed it to handle hurricane-force winds by flexing without breaking.*

there are no Dumb Questions

Q: Why are they called *unit* tests?

A: The term "unit test" is a generic term that applies to many different languages, not just C#. It comes from the idea that your code is divided into discrete units, or small building blocks. Different languages have different units; in C#, the basic unit of code is a class.

So from that perspective, the name "unit test" makes sense: you write tests for the units of code, or in our case, on a class-by-class basis.

Q: I created two projects in a single solution. How does that work, exactly?

A: When you start a new C# project in Visual Studio, it creates a solution and adds a project to it. All of the solutions that you created so far in the book had a single project—until the unit test project. A solution can actually contain many projects. We used a separate project to keep the unit tests separate from the code that they're testing. You can also add multiple Console App or .NET MAUI projects—a solution can contain combinations of different project types.

Q: If a solution has multiple Console or .NET MAUI projects, how does the IDE know which one to run?

A: In Visual Studio, one of the project names is boldfaced. The IDE calls that its **Startup Project**. You can right-click on any project in the solution and tell it to use it as the Startup Project instead. Then the next time you press the Run button in the toolbar, that project will start. In VSCode, you can run any project by opening any of its C# code files and clicking the run button.

Q: I'm still not clear on the internal access modifier. Can you explain it?

A: When you mark a class or interface internal, that means it can only be accessed from inside that project. If you don't use an access modifier, a class or interface defaults to internal. That's why you had to make sure your classes were marked public—otherwise, the unit tests wouldn't be able to see them. Also, *be careful with access modifiers*: while classes and interfaces default to internal if you leave off the access modifier, class members like methods, fields, and properties default to private.

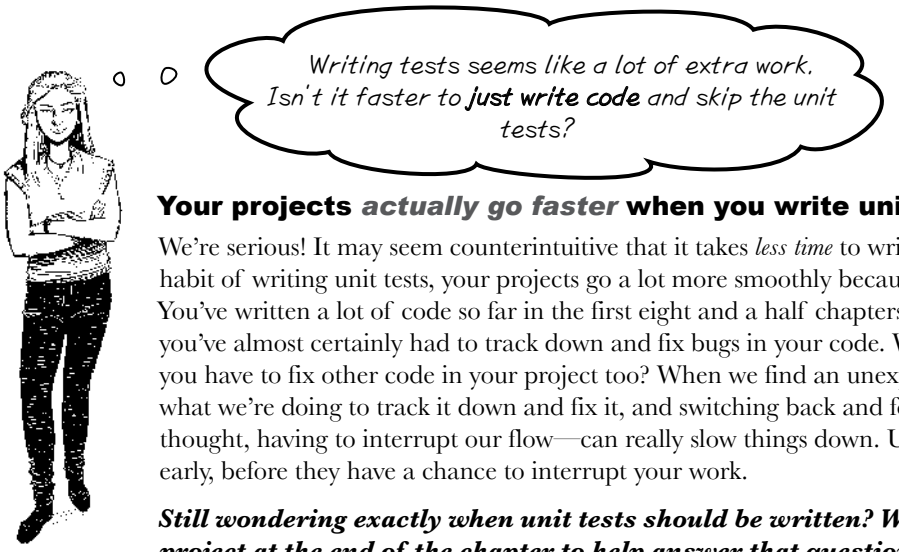
Q: If my unit tests are in a separate project from the code that they're testing, how can they access private methods?

A: They don't. Unit tests access whatever part of the unit is visible to the rest of the code—so for your C# classes, that means the public methods and fields—and use them to make sure the unit works. Unit tests are typically supposed to be **opaque box tests**, which means that they only check the methods that they can see (as opposed to clear box tests, where you can "see" the internals of the thing that you're testing).

Q: Do all of my tests need to pass? Is it OK if some of them fail?

A: Yes, all of your tests need to pass. And no, it's not OK if they fail. Think of it this way—is it OK if some of your code doesn't work? Of course not! So if your test fails, then either the code has a bug or the test does. Either way, you should fix it so it passes.

Failed tests are a good thing! They tell you that you've got a problem to sleuth out.



Your projects *actually go faster* when you write unit tests.

We're serious! It may seem counterintuitive that it takes *less time* to write *more code*, but if you're in the habit of writing unit tests, your projects go a lot more smoothly because you find and fix bugs early. You've written a lot of code so far in the first eight and a half chapters of this book, which means you've almost certainly had to track down and fix bugs in your code. When you fixed those bugs, did you have to fix other code in your project too? When we find an unexpected bug, we often have to stop what we're doing to track it down and fix it, and switching back and forth like that—losing our train of thought, having to interrupt our flow—can really slow things down. Unit tests help you find those bugs early, before they have a chance to interrupt your work.

Still wondering exactly when unit tests should be written? We included a downloadable project at the end of the chapter to help answer that question.

Sens-AI

Ask an AI to help write unit tests for your classes

We've learned a few things about AI chatbots so far. We know that they're good at reading code and adding comments to explain what that code does. You saw an example of that when you asked an AI to add comments to Owen's ability score calculator in Chapter 4. You've probably also noticed that they're good at taking real-world concepts and turning them into code—for example, it may have recognized that the app had to do with role-playing games.

You also learned about asking AI chatbots to change the way they answer future questions, and how they're good at remembering previous questions and answers when you give them follow-up prompts. Let's take advantage of those capabilities and create a strategy for asking an AI chatbot to help you to write unit tests.

Start with a prompt that tells the AI chatbot how to answer

In Chapters 7 and 8, you experimented with personas by giving the AI specific instructions for how to answer your prompts. Now you'll do something similar for creating unit tests. Start by telling the AI that you'll ask it to write unit tests using MSTest, and instruct it to include comments that show how it's using the Arrange-Act-Assert pattern:

> *I will be asking you to write unit tests in C# using MSTest. In your answers, use comments to indicate which parts of the code correspond to the Arrange, Act, and Assert parts of the unit test.*

Give the AI chatbot the code that you'll be testing

Follow up with a prompt that provides all of the code you'll ask it to write tests for. Some chatbots may limit the prompt length. If you run out of room to paste code, split it up into multiple prompts:

> *I will be asking you to create unit tests for the following code:*
>
> *(paste the complete contents of the code to write tests for -- for example, to test Jimmy's comic analyzer app, paste all of the code for Comic, ComicAnalyzer, Critics, PriceRange, Review, and Reviews classes)*

Ask the AI chatbot to test specific cases or conditions—especially edge cases

Now that you've given the AI the context that it needs, ask it to generate tests. But the AI can't do all the work for you! Your job is to figure out the **specific situations** or cases that you want to test. Can the code handle very large or very small values? Negative numbers? Empty collections, or ones with repeated values? Null values? This is the fun part!

Here's an example of a prompt to generate a unit test to write from Jimmy's comic app.

> *Write a unit test for the ComicAnalyzer.GetReviews method to verify that it can accept an empty sequence of comics.*

It can be really useful to have more than one test for a method, to make sure all of its code works.

Research any assertions that you don't recognize

One AI chatbot we asked gave us this code, which returns the reviews in a variable called result and calls result.Any:

```
 Assert.IsFalse(result.Any(), "No reviews should be returned for empty sequence of comics");
```

`Any` is a LINQ method that we haven't talked about in this chapter, so we asked it a follow-up question:

> *Why did the test call the result.Any method? How does that method work?*

Use the => operator to create lambda expressions

We left you hanging back at the beginning of the chapter. Remember that mysterious line we asked you to add to the Comic class? Here it is again:

```
public override string ToString() => $"{Name} (Issue #{Issue})";
```

You've been using that ToString method throughout the chapter—you know it works. What would you do if we asked you to rewrite that method the way you've been writing methods so far? You might write something like this:

```
public override string ToString() {
    return $"{Name} (Issue #{Issue})";
}
```

And you'd basically be right. So what's going on? What exactly is that => operator?

The => operator that you used in the ToString method is the **lambda operator**. You can use => to define a **lambda expression**, or an ***anonymous function*** *defined within a single statement*. Lambda expressions look like this:

```
(input-parameters) => expression;
```

There are two parts to a lambda expression:

- The `input-parameters` part is a list of parameters, just like you'd use when you declare a method. If there's only one parameter, you can leave off the parentheses.
- The `expression` is any C# expression: it can be an interpolated string, a statement that uses an operator, a method call—pretty much anything you would put in a statement.

Lambda expressions may look a little weird at first, but they're just another way of using the ***same familiar C# expressions*** that you've been using throughout the book—just like the Comic.ToString method, which works the same way whether or not you use a lambda expression.

If that ToString method works ***the same way*** *whether or not you use a lambda expression, isn't that a lot like* ***refactoring****?*

Yes! You can use lambda expressions to refactor many methods and properties.

You've written a lot of methods throughout this book that contain just a single statement. You could refactor most of them to use lambda expressions instead. In many cases, that could make your code easier to read and understand. Lambdas give you options—you can decide when using them improves your code.

A Lambda Test Drive

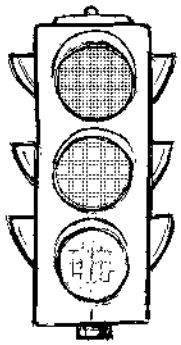
Let's kick the tires on lambda expressions, which give us a whole new way to write methods, including ones that return values or take parameters.

1 **Create a new console app and add this code to the *Program.cs* file.**

Remember, you can include classes at the bottom of your *Program.cs* file—you don't have to put them into their own *.cs* files—which is convenient for small examples like this.

```
new Nest(8).WriteLines();

class Nest
{
    public Nest(int count) {
        Count = count;
    }

    public int Count { get; private set; }

    public string Chicks = GetChicks();

    public static string GetChicks() {
        return $"The nest has {Random.Shared.Next(1, 12)} chicks";
    }

    public void WriteLines() {
        for (int i = 0; i < Count; i++)
            Console.WriteLine(Chicks);
    }
}
```

We're using a regular constructor and not a primary constructor to show how you can use constructors with lambda expressions.



Run it a few times. Each time it prints eight identical lines with a random number: `The nest has 6 chicks`

2 **Use the lambda operator => to refactor the GetChicks method and the constructor.**

Remove the **{** brackets **}** and replace the **`return`** keyword with **=>** in the GetChicks method.

```
public static string GetChicks() => $"The nest has {Random.Shared.Next(1, 12)} chicks";
```

Run your app again—it prints the same output that it did before. Now do the same for the constructor:

```
public Nest(int count) => Count = count;
```

Your app still works the same. Your methods are converted to lambdas.

3 **Modify the Chicks field to use the lambda operator.**

Modify the Chicks field to change the equals = to a lambda => like this:

```
public string Chicks => GetChicks();
```

Run your app again. Now instead of printing the ***same*** line eight times, it prints eight ***different*** lines. That's because replacing = with the lambda operator **converted your Chicks field into a lambda expression** that gets executed every time Console.WriteLine accesses the Chicks field.

Changing Chicks to a lambda expression causes it to get evaluated every time it's used.

```
The nest has 9 chicks
The nest has 4 chicks
The nest has 11 chicks
The nest has 6 chicks
The nest has 6 chicks
The nest has 9 chicks
The nest has 11 chicks
The nest has 10 chicks
The nest has 1 chicks
The nest has 7 chicks
```

Refactor a clown with lambdas

Do this!

Back in Chapter 7, you created an IClown interface with two members:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

The IClown interface from Chapter 7 has two members: one property and one method.

And you modified this class to implement that interface:

```
class TallGuy {
    public string? Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine($"My name is {Name} and I'm {Height} inches tall.");
    }
}
```

IDE Tip: Implement Interface

When a class implements an interface, the Quick Actions menu's **"Implement interface"** option tells the IDE to add any missing interface members.

Let's do that same thing again—but this time we'll use lambdas. **Create a new Console App project** and add the IClown interface and TallGuy class. Then modify TallGuy to implement IClown:

```
class TallGuy : IClown {
```

Now open the Quick Actions menu and choose **"Implement interface."** The IDE fills in all of the interface members, having them throw NotImplementedExceptions just like it does when you use Generate Method.

```
    public string FunnyThingIHave => throw new NotImplementedException();

    public void Honk()
    {
        throw new NotImplementedException();
    }
```

When you asked the IDE to implement the IClown interface for you, it used the => operator to create a lambda to implement the property.

Let's refactor these methods so they do the same thing as before, but now use lambda expressions:

```
    public string FunnyThingIHave => "big red shoes";

    public void Honk() => Console.WriteLine("Honk honk!");
```

The IDE created a method body when it added the interface members, but you can replace it with a lambda expression. When a class member has a lambda as a body, it's called an *expression-bodied member*.

Now add the same top-level statements that you used back in Chapter 7:

```
TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
tallGuy.TalkAboutYourself();
Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
tallGuy.Honk();
```

That mysterious ToString method at the beginning of the chapter was an expression-bodied member.

Run your app. The TallGuy class works just like it did in Chapter 7, but now that we've refactored its members to use lambda expressions, it's more compact.

We think the new and improved TallGuy class is easier to read. Do you?

Sharpen your pencil

Here are the NectarCollector class from the Beehive Management System project in Chapter 6 and the ScaryClown class from Chapter 7. Your job is to **refactor some of the members of these classes using the lambda operator (=>)**. Write down the refactored methods.

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}
```

Refactor the CostPerShift property as a lambda expression-bodied member:

.....

```
class ScaryClown : FunnyClown, IScaryClown
{
    private int scaryThingCount;

    public ScaryClown(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }

    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }

    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}!");
    }
}
```

Refactor the ScaryThingIHave property as a lambda expression-bodied member:

.....

Refactor the ScareLittleChildren method as a lambda expression:

.....

Sharpen your pencil Solution

Refactor the CostPerShift property as a lambda expression-bodied member:

```
public override float CostPerShift =>1.95f;
```

Refactor the ScaryThingIHave property as a lambda expression-bodied member:

```
public string ScaryThingIHave { get => $"{scaryThingCount} spiders"; }
```

Refactor the ScareLittleChildren method as a lambda expression:

```
public void ScareLittleChildren() => Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}");
```

there are no Dumb Questions

Q: I'm still not clear on what happened in the Lambda Test Drive. Why did the app change when I modified the Chicks field to use the lambda operator?

A: When you created the Nest class at the start of the Lambda Test Drive, you created a Chicks field and initialized it by calling the GetChicks method:

```
public string Chicks = GetChicks();
```

This works exactly the way you'd expect: when the Nest class is instantiated, it initializes the Chicks field by calling the GetChicks method and setting the field to whatever value is returned by that method. When the WriteLines method executes `Console.WriteLine(Chicks)` inside a `for` loop, it prints the value of the field each time.

Everything changed when you replaced the equals sign (=) with a lambda operator (=>). The new version of Chicks is still a field. But now that you changed the operator, when you get the value of the field, **it executes a lambda expression**. So when you access Chicks, instead of just returning a value, it executes that expression and calls the GetChicks method.

Q: So Chicks isn't a field anymore? It's more like a method that gets executed each time you access it?

A: That's right. We said earlier that a lambda expression is an "**anonymous function** defined within a single statement"—and that's exactly what we just saw happening with the Chicks field.

An "anonymous function" is an expression that represents an "in-line" method definition—that's another way of saying that it's a method that doesn't have a name, and instead is just defined by whatever comes after the => operator.

So when you replaced the = with a lambda operator:

```
public string Chicks => GetChicks();
```

you turned Chicks into a method that calls GetChicks() every time it's accessed.

You can use the => operator to create a field or a property that executes a lambda expression.

Use the ?: operator to make your lambdas make choices

What if you want your lambdas to do... more? It would be great if they could make decisions…and that's where the **conditional operator** (which some people call the **ternary operator**) comes in. It works like this:

```
condition ? consequent : alternative;
```

which may look a little weird at first, so let's have a look at an example. First of all, the ?: operator isn't unique to lambdas—you can use it anywhere. Take this **`if`** statement from the AbilityScoreCalculator class in Chapter 4:

```
if (added < Minimum)
    Score = Minimum;
else
    Score = added;
```

The ?: expression checks the condition (added < Minimum).

If it's true, the expression returns the value Minimum.

Otherwise it returns added.

We can refactor it using the ?: operator like this: **`Score = (added < Minimum) ? Minimum : added;`**

Notice how we set Score equal to the results of the ?: expression. The ?: expression **returns a value**: it checks the *condition* (added < Minimum), and then it either returns the *consequent* (Minimum) or the *alternative* (added).

When you have a method that looks like that **`if/else`** statement, you can **use ?: to refactor it as a lambda**. For example, take this method from the PaintballGun class in Chapter 5:

```
public void Reload()
{
   if (balls > MAGAZINE_SIZE)
      BallsLoaded = MAGAZINE_SIZE;
   else
      BallsLoaded = balls;
}
```

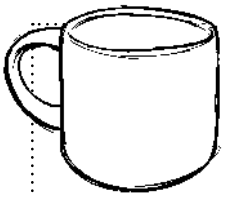
This "if" conditional (balls > MAGAZINE_SIZE) executes the then statement (BallsLoaded = MAGAZINE_SIZE) if it's true, or the else statement (BallsLoaded = balls) if it's not.

We converted the "if" to an expression-bodied member that uses ?:—both the consequent and alternative both return a value—and used that to set the BallsLoaded property.

Let's rewrite that as a more concise lambda expression:

```
public void Reload() => BallsLoaded = balls > MAGAZINE_SIZE ? MAGAZINE_SIZE : balls;
```

Notice the slight change—in the **`if/else`** version, the BallsLoaded property was set inside the **`then`** and **`else`** statements. We changed this to use a conditional operator that checked balls against MAGAZINE_SIZE and returned the correct value, and used that return value to set the BallsLoaded property.



Relax

There's an easy way to remember how the conditional operator works.

A lot of people have trouble remembering the order of the question mark and colon in the ?: ternary operator. Luckily, there's an easy way to remember it.

The conditional operator is like asking a question, and you always ask a question before you find out the answer. So just ask yourself:

is this condition true ? yes : no

and you'll know that the ? appears before the : in your expression.

Here's a fun fact—we learned this tip from Microsoft's great documentation page for the ?: operator: *https://docs.microsoft.com/dotnet/csharp/language-reference/operators/conditional-operator*

LINQ queries are made up of methods

You've been building LINQ queries with keywords like **`select`**, **`orderby`**, and **`join`**. But what exactly is happening when you use those keywords? Let's use Visual Studio to explore them and find out what's going on.

Add this small LINQ query to any C# app, then hover over the **`select`** keyword in the code:

```
var array = new[] { 1, 2, 3, 4 };
var result = from i in array select i * 2;
```

This looks like the same pop-up window that you get when you hover over a method.

```
(extension) IEnumerable<int> IEnumerable<int>.Select<int, int>(Func<int, int> selector)
Projects each element of a sequence into a new form.

Returns:
  An IEnumerable<out T> whose elements are the result of invoking the transform function
  on each element of source.

Exceptions:
  ArgumentNullException
```

The method takes a Func<int, int> parameter.

This means **`select`** ***is really a method***.

You're looking at a method declaration—and you've seen a lot of method declarations before. This one is a little bit longer, but if we look at it piece by piece, we can make sense of it:

```
IEnumerable<int> IEnumerable<int>.Select<int, int>(Func<int, int> selector)
```

- The first part of the declaration is **`IEnumerable<int>`**—that's the return type.
- The next part is **`IEnumerable<int>.`**—which means this is a static method for the type IEnumerable<int>.
- After that comes **`Select<int, int>`**—the important part here is that the method name is Select.
- It ends with **`(Func<int, int> selector)`**—it takes a single parameter of type **`Func<int, int>`**.

You can use lambda expressions with methods that take a Func parameter

When a method takes a **`Func<int, int>`** parameter, you can **call it with a lambda expression** that takes an int parameter and returns an int value. So you could refactor the select query like this:

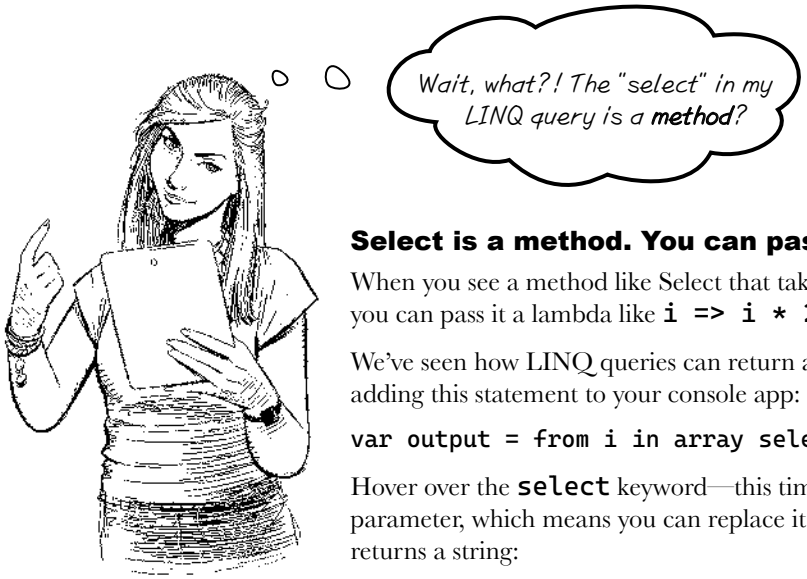
```
var array = new[] { 1, 2, 3, 4 };
var result = array.Select(i => i * 2);
```

i => i * 2 is a lambda expression that takes an int value, multiplies it by 2, and returns an int.

Go ahead—try that yourself in a console app. Add a **`foreach`** statement to print the output:

```
foreach (var i in result)
    Console.WriteLine(i);
```

When you print the results of the refactored query, you'll get the sequence { 2, 4, 6, 8 }—which is exactly the same result as you got with the LINQ query syntax before you refactored it to call the Select method instead.



Select is a method. You can pass it a lambda expression.

When you see a method like Select that takes a `Func<int, int>` parameter, you can pass it a lambda like `i => i * 2` that takes an int and returns an int.

We've seen how LINQ queries can return any type (even anonymous types). Try adding this statement to your console app:

```
var output = from i in array select $"The number is {i}";
```

Hover over the `select` keyword—this time it takes a `Func<int, string>` parameter, which means you can replace it with a lambda that takes an int and returns a string:

```
var output = array.Select(i => $"The number is {i}");
```

Let's use Visual Studio's refactoring tools to explore Func a little more. Use your mouse to select `i => i * 2`:

```
var array = new[] { 1, 2, 3, 4 };
var result = array.Select(i => i * 2);

foreach (var i in result)
    Console.WriteLine(i);
```

Quick Actions and Refactorings...	Ctrl+.
Rename...	F2
Remove and Sort Usings	Ctrl+R, Ctrl+G

Right-click and choose Quick Actions and Refactorings (Ctrl+. or ⌘+.), then choose the **Extract Local Function** quick action. You'll be asked for a method name (just like when you rename a method)—call it TimesTwo. You'll see this:

```
var array = new[] { 1, 2, 3, 4 };
var result = array.Select(TimesTwo());

foreach (var i in result)
    Console.WriteLine(i);

static Func<int, int> TimesTwo()
{
    return i => i * 2;
}
```

The Select method takes a Func<int, int>. Visual Studio replaced your lambda with this method call.

The TimesTwo method it created returns a Func<int, int>. So because Select takes a Func<int, int>, it can call the TimesTwo method and use the value that it returns.

So let's put it all together. The lambda i => i * 2 takes an int and returns an int, so it can be used anywhere you see Func<int, int>, including the return value of the TimesTwo method. Since the Select method takes a Func<int, int> parameter, it can use the TimesTwo method return value.

LINQ declarative syntax can be refactored into chained methods

Here's the LINQ query we used earlier, with one extra clause—we added **`orderby i descending`** before the **`select`**:

```
var array = new[] { 1, 2, 3, 4 };
var result = from i in array
             orderby i descending
             select i * 2;

foreach (var i in result)
    Console.WriteLine(i);
```

Do this!

Create a new console app and **add this code**. Hover your mouse over the **`descending`** keyword (not orderby):

```
(extension) IOrderedEnumerable<int> IEnumerable<int>.OrderByDescending<int, int>(Func<int, int> keySelector)
Sorts the elements of a sequence in descending order according to a key.

Returns:
 An IOrderedEnumerable<out TElement> whose elements are sorted in descending order according to a key.

Exceptions:
 ArgumentNullException
```

Just like before, the IntelliSense window shows you useful information:

- The **`descending`** keyword is equivalent to a method called **`OrderByDescending`**.
- The OrderByDescending method takes a **`Func<int, int>`** parameter.
- The method returns an **`IOrderedEnumerable<int>`** value.

Take a closer look at that return value. Back in Chapter 7, we learned how one interface can extend another, like how IScaryClown extended IClown. In this case, IOrderedEnumerable extends the IEnumerable interface. That means you can call any LINQ method on its return value. That means you can **chain LINQ methods**.

Let's try that now. Replace the **`orderby i descending`** clause with the method call OrderByDescending(i => i) and then replace the **`select`** clause with the same method call you used before:

```
var result = array
    .OrderByDescending(i => i)
    .Select(i => i * 2);

foreach (var i in result)
    Console.WriteLine(i);
```

The orderby descending clause orders by i, so you'll use the lambda i => i, which returns the same int value that was passed to it.

Run your app again. It still does the same thing, but now you're using chained method calls to do exactly the same thing the LINQ declarative query syntax did. You can do the same thing with any LINQ query and replace the declarative syntax with chained LINQ method calls..

You can replace any LINQ query that uses the declarative syntax with a series of chained LINQ method calls.

Exercise

Here's a LINQ query that uses declarative syntax. Can you figure out how to refactor it into a set of chained LINQ methods? Start with **result**, then chain the Where and OrderBy methods to produce the same sequence.

```
int[] values = [ 0, 12, 44, 36, 92, 54, 13, 8 ];
IEnumerable<int> result =
            from v in values
            where v < 37
            orderby -v
            select v;
```

A sequence of ints has a LINQ method called OrderBy with a lambda that takes an int and returns an int. It works just like the comparer methods we saw in Chapter 8.

Step 1: Use IntelliSense to figure out the method for the `orderby` clause.

Hover over the **orderby** keyword and take a look at its parameter:

```
(extension) IOrderedEnumerable<int> IEnumerable<int>.OrderBy<int, int>(Func<int, int> keySelector)
Sorts the elements of a sequence in ascending order according to a key.

Returns:
  An IOrderedEnumerable<out TElement> whose elements are sorted according to a key.
```

When you use an **orderby** clause in a LINQ query, it calls a LINQ OrderBy method that sorts the sequence. In this case, we can pass it a lambda expression with an int parameter that **returns the sort key**, or any value (which must implement IComparer) that it can use to sort the results.

The LINQ Where method uses a lambda that takes a member of a sequence and returns true if it should be kept, or false if it should be removed.

Step 2: Use IntelliSense to figure out the method for the `where` clause.

Now hover over the **where** keyword in the LINQ query:

```
(extension) IEnumerable<int> IEnumerable<int>.Where<int>(Func<int, bool> predicate)
Filters a sequence of values based on a predicate.

Returns:
  An IEnumerable<out T> that contains elements from the input sequence that satisfy the condition.
```

The **where** clause in a LINQ query calls a LINQ Where method that can use a lambda that returns a Boolean. ***The Where method calls that lambda for each element in the sequence.*** If the lambda returns true, the element is included in the results. If the lambda returns false, the element is removed.

Step 3: Finish with the `select` clause.

You've seen an example of how the **select** clause works. Finish the method chain with a Select method.

It's OK if Func seems a little weird at first. For now, just think about it like this: any time you see a Func in a LINQ method, it just means you can use a lambda. You'll learn a lot more about Func in the downloadable chapter on events and delegates, which is available as a PDF on our GitHub page: *https://github.com/head-first-csharp/fifth-edition*

LINQ Methods Up Close

LINQ queries can be rewritten as a series of chained LINQ methods—and many of those methods can use lambda expressions to determine the sequence that they produce.

Here's the exercise solution—the LINQ query can be refactored into chained method calls like this:

```
var result = values.Where(v => v < 37).OrderBy(v => -v);
```

Exercise Solution

Let's take a closer look at how the LINQ query is turned into chained methods:

```
IEnumerable<int> result =
```
Use var to declare the variable → `var result =`

```
    from v in values
```
Start with the values sequence → `values`

```
    where v < 37
```
Call Where with a lambda that includes values under 37 → `.Where(v => v < 37)`

```
    orderby -v
```
Call OrderBy with a lambda that negates the value → `.OrderBy(v => -v);`

```
    select v;
```
No need for a .Select method because the `select` clause doesn't modify the value

Use the OrderByDescending method where you'd use the `descending` keyword in a LINQ query

Remember how you used the **`descending`** keyword to change the **`orderby`** clause in the query? There's an equivalent LINQ method, OrderByDescending, that does exactly the same thing:

```
var result = values.Where(v => v < 37).OrderByDescending(v => v);
```

Notice how we're using the lambda expression `v => v`—that's a lambda that will always return whatever is passed to it (sometimes referred to an *identity function*). So, `OrderByDescending(v => v)` reverses a sequence.

Use the GroupBy method to create group queries from chained methods

We saw this group query earlier in the chapter:

```
var grouped =
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
```

Should you use LINQ declarative query syntax or chained methods? They both accomplish the same thing. Sometimes one way leads to clearer code, sometimes the other way does, so it's valuable to know how to use both.

Hover over **`group`**, and you'll see that it calls the LINQ GroupBy method, which returns the same type we saw earlier in the chapter. You can use a lambda to group by the card's suit: `card => card.Suit`

And then another lambda to order the groups by key: `group => group.Key`

```
(extension) IEnumerable<IGrouping<Suits, Card>> IEnumerable<Card>
.GroupBy<Card, Suits>(Func<Card, Suits> keySelector)
Groups the elements of a sequence according to a specified key selector function.
```

Here's the LINQ query refactored into chained GroupBy and OrderByDescending methods:

```
var grouped =
    deck.GroupBy(card => card.Suit)
    .OrderByDescending(group => group.Key);
```

Here's another one of those complicated-looking method declarations. Can you spot the Func<Card, Suits> parameter?

Try going back to the app earlier in the chapter where you used that query and replace it with the chained methods. You'll see exactly the same output. It's up to you to decide which version of your code is clearer and easier to read.

Use the => operator to create switch expressions

You've been using **switch** statements since Chapter 6 to check a variable against several options. It's a really useful tool...but have you noticed its limitations? For example, try adding a case that tests against a variable:

```
case myVariable:
```

You'll get a C# compiler error: *A constant value is expected.* That's because you can only use constant values—like literals and variables defined with the **const** keyword—in the **switch** statements that you've been using.

But that all changes with the **=>** operator, which lets you create **switch expressions**. They're similar to the **switch** statements that you've been using, but they're *expressions* that return a value. A switch expression starts with a value to check and the **switch** keyword followed by a series of *switch arms* in curly brackets separated by commas. Each switch arm uses the => operator to check the value against an expression. If the first arm doesn't match, it moves on to the next one, returning the value for the matching arm.

```
var returnValue = valueToCheck switch
{
    pattern1 => returnValue1,
    pattern2 => returnValue2,
    ...
    _ => defaultReturnValue,
}
```

A switch expression starts with a value to check followed by the "switch" keyword.

The body of the switch expression is a series of switch arms that use the => operator to check valueToCheck and return a value if it matches a pattern.

Switch expressions must be *exhaustive*, which means their patterns must match every possible value. The _ pattern will match any value that hasn't been matched by any other arm.

Let's say you're working on a card game that needs to assign a certain score based on suit, where spades are worth 6, hearts are worth 4, and other cards are worth 2. You could write a **switch** statement like this:

```
var score = 0;
switch (card.Suit)
{
     case Suits.Spades:
        score = 6;
        break;
     case Suits.Hearts:
        score = 4;
        break;
     default:
        score = 2;
        break;
}
```

Every case in this switch statement sets the score variable. That makes it a great candidate for switch expressions.

The whole goal of this **switch** statement is to use the cases to set the **score** variable—and a lot of our **switch** statements work that way. We can use the => operator to create a switch expression that does the same thing:

```
var score = card.Suit switch
{
    Suits.Spades => 6,
    Suits.Hearts => 4,
    _ => 2,
};
```

This switch expression checks card.Suit—if it's equal to Suits.Spades the expression returns 6, if it's equal to Suits.Hearts it returns 4, and for any other value it returns 2.

Sharpen your pencil

This console app uses the Suit, Value, and Deck classes that you used earlier in the chapter in your CardLinq project and writes six lines to the console. Your job is to **write down the output of the program**. When you're done, add the program to a console app to check your answer.

```
using CardLinq;

static string Output(Suits suit, int number) => $"Suit is {suit}, number is {number}";

var deck = new Deck();
var processedCards = deck
    .Take(3)
    .Concat(deck.TakeLast(3))
    .OrderByDescending(card => card)
    .Select(card => card.Value switch
    {
        Values.King => Output(card.Suit, 7),
        Values.Ace => $"It's an ace! {card.Suit}",
        Values.Jack => Output((Suits)card.Suit - 1, 9),
        Values.Two => Output(card.Suit, 18),
        _ => card.ToString(),
    }) ;

foreach(var output in processedCards)
{
    Console.WriteLine(output);
}
```

This lambda expression takes two parameters, a Suit and an int, and returns an interpolated string.

These LINQ methods are just like the ones you saw at the beginning of the chapter.

You can use OrderByDescending because you made your Card class implement IComparable<Card> earlier in the chapter.

The Select method uses a switch expression to check the card's value and generate a string.

Write down the output of the program. **We won't give you a solution**—add the code to a console app for the answer. When you add the code to the console app, make sure you add the Card, Deck, and CardComparerByValue classes and Suits and Values enums from the CardLinq project to your new console app.

This is a *serious lambda challenge*! There's a lot going on here—you're using lambda expressions, a switch expression, LINQ methods, enum casting, chained methods, and more. Really take your time and figure out how this code works before writing down your solution, and then run the program. If your solution didn't match the output, it's a great opportunity to sleuth out why it worked differently than you expected.

Exercise

Use everything you've learned about lambda expressions, switch expressions, and LINQ methods to refactor the ComicAnalyzer class and top-level statements, using your unit tests to make sure your code still works.

Replace the LINQ queries in ComicAnalyzer

ComicAnalyzer has two LINQ queries:

- The GroupComicsByPrice method has a LINQ query that uses the `group` keyword to group comics by price.
- The GetReviews method has a LINQ query that uses the `join` keyword to join a sequence of Comic objects to a Dictionary of issue prices.

Modify the LINQ queries in these methods to use the LINQ OrderBy, GroupBy, Select, and Join methods. There's one catch: ***we haven't shown you the Join method yet!*** But we showed you examples of how to use the IDE to explore LINQ methods. The Join method is a little more complex—but we'll help you break it down. It takes four parameters:

```
sequence.Join(sequence to join,
          lambda expression for the 'on' part of the join,
          lambda expression for the 'equals' part of the join,
          lambda expression that takes two parameters and returns the 'select' output);
```

Look closely at the "on" and "equals" parts of the LINQ query to come up with the first two lambdas. The Join will be the last method in the chain. *Here's a hint*—the last parameter's lambda starts like this: `(comic, review) =>`

Once both unit tests pass, then you're done refactoring the ComicAnalyzer class.

Replace the `switch` statement in the top-level statements with a switch expression

The top-level statements have a `switch` statement that calls private methods and assigns their return values to the `done` variable. Replace this with a switch expression with three switch arms. You can test it by running the app—if you can press the correct key and see the right output, you're done.

This exercise is about learning to use unit tests to *safely* refactor your code

Behind the Scenes

Refactoring can be messy, even nerve-racking business. You're taking code that works and making changes to improve its structure, readability, and reusability. When you're modifying your code, it's really easy to accidentally mess it up so it doesn't quite work right anymore—and sometimes the bugs you introduce can be subtle and difficult to track down, or even detect. That's where unit tests can help. One of the most important ways that developers use unit tests is to make refactoring a much safer activity. Here's how it works:

- Before you start refactoring, write tests that make sure your code works—like the tests you added earlier in the chapter to validate the ComicAnalyzer class.
- When you're refactoring a class, just run the tests for that class as you make the changes. That gives you a much shorter feedback loop for developing—you can do your normal debugging, but it goes a lot faster because you're executing the code in the class directly (instead of running your app and using it to run the code that uses the class).
- When you're refactoring a method, you can even start by just running the specific test or tests that execute that method. Then, once it works, you can run the entire suite to make sure you didn't break anything else.
- If a test fails, don't feel bad—that's actually good news! It's telling you something's broken, and now you can fix it.

Exercise Solution

Use everything you've learned about lambda expressions, switch expressions, and LINQ methods to refactor the ComicAnalyzer class and top-level statements, using your unit tests to make sure your code still works.

Here are the refactored GroupComicsByPrice and GetReviews methods from the ComicAnalyzer class:

```
public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
                  IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
{
   var grouped =
       comics
    .OrderBy(comic => prices[comic.Issue])
    .GroupBy(comic => CalculatePriceRange(comic, prices));

  return grouped;
}
```

Compare the OrderBy and GroupBy lambdas with the orderby and group...by clauses in the LINQ query. They're almost identical.

```
public static IEnumerable<string> GetReviews(
                 IEnumerable<Comic> comics, IEnumerable<Review> reviews)
{
  var joined =
    comics
    .OrderBy(comic => comic.Issue)
    .Join(
      reviews,
      comic => comic.Issue,
      review => review.Issue,
      (comic, review) =>
           $"{review.Critic} rated #{comic.Issue} '{comic.Name}' {review.Score:0.00}");

  return joined;
}
```

The join query starts "join reviews" so the first argument passed to the Join method is reviews.

Compare the middle two arguments passed to the Join method against the "on" and "equals" parts of the join query: on comic.Issue equals review.Issue."

This last lambda is called with every matched comic and review pair from the two joined sequences and returns the string to include in the output.

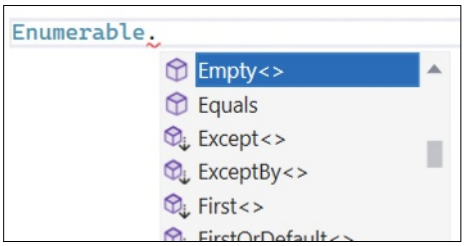
Here are the refactored top-level statements that use a switch expression instead of a `switch` statement:

```
var done = false;
while (!done)
{
  Console.WriteLine(
      "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
  done = Console.ReadKey(true).KeyChar.ToString().ToUpper() switch
  {
     "G" => GroupComicsByPrice(),
     "R" => GetReviews(),
     _ => true,
  };
}
```

The switch expression is a lot more compact than the equivalent switch statement. Not all switch statements can be refactored into switch expressions—this one could be, because each of its cases sets the same variable (done) to a value.

Explore the Enumerable class

We've been using sequences for a while. We know they work with **`foreach`** loops and LINQ. But what, exactly, makes sequences tick? Let's take a deeper dive to find out. We'll start with the **Enumerable class**—specifically, with its three static methods: Range, Empty, and Repeat. You already saw the Enumerable.Range method earlier in the chapter. Let's use the IDE to discover how the other two methods work. Type **`Enumerable.`** and then hover over Range, Empty, and Repeat in the IntelliSense pop-up to see their declarations and comments.



IEnumerable<TResult> Enumerable.Empty<TResult>()
Returns an empty IEnumerable<out T> that has the specified type argument.

IEnumerable<TResult> Enumerable.Repeat<TResult>(TResult element, int count)
Generates a sequence that contains one repeated value.

Enumerable.Empty creates an empty sequence of any type

Sometimes you need to pass an empty sequence to a method that takes an IEnumerable<T> (for example, in a unit test). The **Enumerable.Empty method** comes in handy in these cases:

```
var emptyInts = Enumerable.Empty<int>(); // an empty sequence of ints
var emptyComics = Enumerable.Empty<Comic>(); // an empty sequence of Comic
references
```

Enumerable.Repeat repeats a value a specific number of times

Let's say you need a sequence of 100 3s, or 12 "yes" strings, or 83 identical anonymous objects. You'd be surprised at how often that happens! You can use the **Enumerable.Repeat method** for this—it returns a sequence of repeated values:

```
var oneHundredThrees = Enumerable.Repeat(3, 100);
var twelveYesStrings = Enumerable.Repeat("yes", 12);
var eightyThreeObjects = Enumerable.Repeat(
    new { cost = 12.94M, sign = "ONE WAY", isTall = false }, 83);
```

So what exactly is an IEnumerable<T>?

We've been using IEnumerable<T> for a while now. We haven't really answered the question of what an enumerable sequence *actually is*. A really effective way to understand something is to build it ourselves, so let's finish the chapter by **building some sequences** from the ground up.

Brain Power

If you had to design an IEnumerable<T> interface yourself, what members would you put in it?

Create an enumerable sequence by hand

Do this!

Let's say we have some sports in an enum, and we want to add them to a sequence:

```
enum Sport { Football, Baseball, Basketball, Hockey, Boxing, Rugby, Fencing }
```

Obviously, we could create a new List<Sport> and use a collection initializer to populate it. But for the sake of exploring how sequences work, we'll build one manually.

Create a new console app. Add the Sport enum, then add a new class called ManualSportSequence. Make it implement the IEnumerable<Sport> interface—make sure you use the **"Implement interface" Quick Action** to do it. You'll see that the interface has two members that return an IEnumerator:

```
using System.Collections;

class ManualSportSequence : IEnumerable<Sport>
{
    public IEnumerator<Sport> GetEnumerator()
    {
        return new ManualSportEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

When we used the "Implement interface" Quick Actions menu item, it added this using directive because IEnumerator and IEnumerable are in the System.Collections namespace.

IEnumerator<T>
Current
MoveNext Reset Dispose

IEnumerator is an interface that does the job of enumerating a sequence. It has methods to move through each item in the sequence one after another. Its Current returns the current item being enumerated, and its MoveNext method moves to the next element in the sequence, returning false if the sequence has run out. The Reset method resets the sequence back to the beginning. Once you have those methods, you have an enumerable sequence.

So let's implement an IEnumerator<Sport>:

```
class ManualSportEnumerator : IEnumerator<Sport> {
    int current = -1;
    public Sport Current => (Sport)current;

    public void Dispose() {} // You'll meet the Dispose method in Chapter 10

    object IEnumerator.Current => Current;

    public bool MoveNext() {
        var maxEnumValue = Enum.GetValues(typeof(Sport)).Length;
        if ((int)current >= maxEnumValue - 1)
            return false;
        current++;
        return true;
    }
    public void Reset() => current = 0;
}
```

We put this class in the same file as ManualSportSequence so we didn't need a new using directive.

Our manual sport enumerator takes advantage of casting an int to an enum. It uses the static Enum.GetValues method to get the total number of members in the enum, and uses an int to keep track of the index of the current value.

Now you can add a **foreach** loop to the top-level statements to enumerate the sequence:

```
var sports = new ManualSportSequence();
foreach (var sport in sports)
    Console.WriteLine(sport);
```

Use yield return to create your own sequences

C# gives you a much easier way to create enumerable sequences: the **`yield return` statement**. The `yield return` statement is a kind of all-in-one automatic enumerator creator. A good way to understand it is to see an example. Let's use a **multiproject solution**, just to give you a little more practice with that.

Add a new Console App project to your solution—this is just like what you did when you added the MSTest project earlier in the chapter, except this time instead of choosing the project type MSTest, choose the same Console App project type that you've been using for most of the projects in the book. Then right-click on the project under the solution and **choose "Set as startup project."** Now when you launch the debugger in the IDE, it will run the new project. You can also right-click on any project in the solution and run or debug it.

Here's the code for the new console app:

```
IEnumerable<string> SimpleEnumerable() {
    yield return "apples";
    yield return "oranges";
    yield return "bananas";
    yield return "unicorns";
}

foreach (var s in SimpleEnumerable()) Console.WriteLine(s);
```

This method returns an IEnumerable<string> so every yield return returns a string value.

Run the app—it prints four lines: `apples`, `oranges`, `bananas`, and `unicorns`. So how does that work?

Use the debugger to explore yield return

Set a breakpoint on the line with the `foreach` loop and launch the debugger. Then use **Step Into** (F11) to debug the code line by line, right into the iterator:

- Step into the code, and keep stepping into it until you reach the first line of the SimpleEnumerable method.
- Step into that line again. It acts just like a `return` statement, returning control back to the statement that called it—in this case, back to the `foreach` statement, which calls Console.WriteLine to write `apples`.
- Step two more times. Your app will jump back into the SimpleEnumerable method, but ***it skips the first statement in the method*** and goes right to the second line:

```
1  IEnumerable<string> SimpleEnumerable()
2  {
3      yield return "apples";
4      yield return "oranges";   ≤ 4ms elapsed
5      yield return "bananas";
6      yield return "unicorns";
7  }
8
9  foreach (var s in SimpleEnumerable()) Console.WriteLine(s);
```

Every time the foreach loop gets an item from the sequence returned by the SimpleEnumerable method, it jumps back into the method right after the last yield return that was called.

- Keep stepping. The app returns to the `foreach` loop, then back to the ***third line*** of the method, then returns to the `foreach` loop, and goes back to the ***fourth line*** of the method.

So `yield return` makes a method **return an enumerable sequence** by returning the next element in the sequence each time it's called, and keeping track of where it returned from so it can pick up where it left off.

Use yield return to refactor ManualSportSequence

You can create your own IEnumerable<T> by **using `yield return` to implement the GetEnumerator method**. For example, here's a BetterSportSequence class that does exactly the same thing as ManualSportSequence.

Go ahead and **add a new Console App project to your solution** and set it as the default project. Add this new BetterSportSequence class. Modify the top-level statements to create an instance of it and enumerate the sequence. This version is much more compact because it uses `yield return` in its GetEnumerator implementation:

```
using System.Collections;

foreach (var sport in new BetterSportSequence())
    Console.WriteLine(sport);

enum Sport { Football, Baseball, Basketball, Hockey, Boxing, Rugby, Fencing }

class BetterSportSequence : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
        for (int i = 0; i <= maxEnumValue; i++) {
            yield return (Sport)i;
        }
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

You can use yield return to implement the GetEnumerator method in IEnumerable<T> and create your own enumerable sequence.

Add an indexer to BetterSportSequence

You've seen that you can use `yield return` in a method to create an IEnumerator<T>. You can also use it to create a class that implements IEnumerable<T>. One advantage of creating a separate class for your sequence is that you can add an **indexer**. You've already used indexers—any time you use brackets `[]` to retrieve an object from a list, array, or Dictionary (like `myList[3]` or `myDictionary["Steve"]`), you're using an indexer. An indexer is just a method. It looks a lot like a property, except it's got a single named parameter.

The IDE has an ***especially useful code snippet*** to help you add your indexer. Type **`indexer`** followed by two tabs, and the IDE will add the skeleton of an indexer for you automatically.

Here's an indexer for the SportCollection class:

```
public Sport this[int index] {
    get => (Sport)index;
}
```

Calling the indexer with `[3]` returns the value `Hockey`:

```
var sequence = new BetterSportSequence();
Console.WriteLine(sequence[3]);
```

Take a close look when you use the snippet to create the indexer—it lets you set the type. You can define an indexer that takes different types, including strings and even objects. While our indexer only has a getter, you can also include a setter (just like the ones you've used to set items in a List).

Exercise

Create an enumerable class that, when enumerated, returns a sequence of ints that contains all of the powers of 2, starting at 0 and ending with the largest power of 2 that can fit inside an int.

Use `yield return` to create a sequence of powers of 2

Create a class called PowersOfTwo that implements IEnumerable<int>. It should have a `for` loop that starts at 0 and uses `yield return` to return a sequence that contains each power of 2.

The app should write the following output to the console: `1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432 67108864 134217728 268435456 536870912 1073741824`

Return the desired sequence of values

You'll use methods from the static System.Math class in your app to:

- Compute a specific power of 2: `Math.Pow(power, 2)`
- Find the maximum power of 2 that can fit inside an int: `Math.Round(Math.Log(int.MaxValue, 2))`

there are no Dumb Questions

Q: I think I get what's going on with `yield return`, but can you explain again exactly why it jumps right into the middle of a method?

A: When you use `yield return` to create an enumerable sequence, it does something that you haven't seen anywhere else in C#. Normally when your method hits a `return` statement, it causes your program to execute the statement right after the one that called the method. It does the same thing when it's enumerating a sequence created with `yield return`—with one difference: it remembers the last `yield return` statement it executed in the method. Then when it moves to the next item in the sequence, instead of starting at the beginning of the method, your program executes the next statement after the most recent `yield return` that was called. That's why you can build a method that returns an IEnumerable<T> with just a series of `yield return` statements.

Q: When I added a class that implemented IEnumerable<T>, I had to add a MoveNext method and Current property. When I used `yield return`, how was I able to implement that interface without having to implement those two members?

A: When the compiler sees a method with a `yield return` statement that returns an IEnumerable<T>, it automatically adds the MoveNext method and Current property. When it executes, the first `yield return` that it encounters causes it to return the first value to the `foreach` loop. When the `foreach` loop continues (by calling the MoveNext method), it resumes execution with the statement immediately after the last `yield return` that it executed. Its MoveNext method returns false when the enumerator is positioned after the last element in the collection. This may be a little hard to follow on paper, but it's much easier to follow if you load it into the debugger—which is why the first thing we had you do was step through a simple sequence that uses `yield return`.

Sequences are not collections.

Try creating a class that implements ICollection<int> and use the Quick Actions menu to implement its members. A collection not only has to implement the IEnumerable<T> methods, but it also needs additional properties (including Count) and methods (including Add and Clear). That's how you know a collection does a different job than an enumerable sequence.

Exercise Solution

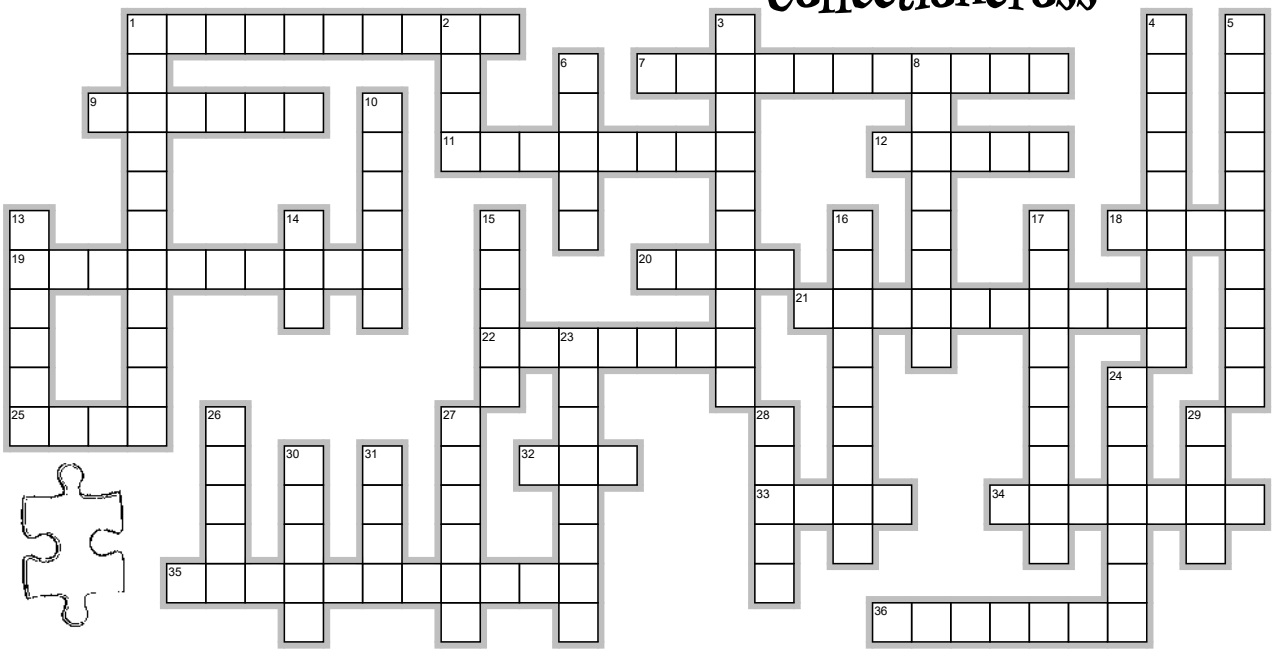
Create an enumerable class that, when enumerated, returns a sequence of ints that contains all of the powers of 2, starting at 0 and ending with the largest power of 2 that can fit inside an int.

```
using System.Collections;

foreach (int i in new PowersOfTwo())
  Console.Write($" {i}");

class PowersOfTwo : IEnumerable<int> {
  public IEnumerator<int> GetEnumerator() {
    var maxPower = Math.Round(Math.Log(int.MaxValue, 2));
    for (int power = 0; power < maxPower; power++)
      yield return (int)Math.Pow(2, power);
  }

  IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Bullet Points

- **Unit tests** are automated tests that help you make sure your code does what it's supposed to do, and help you **safely refactor** your code.
- MSTest is a **unit test framework**, or a set of classes that give you the tools to write unit tests. Visual Studio has tools to execute and see the results of unit tests.
- Unit tests use **assertions** to validate specific behaviors.
- The **`internal` keyword** makes classes in one project inaccessible to another project in a multiproject build.
- Add unit tests to handle edge cases and weird data to make your code more **robust**.
- Use the lambda operator => to define **lambda expressions**, or anonymous functions defined within a single statement that looks like this: `(input-parameters) => expression;`
- When a class implements an interface, the **"Implement interface" option** in the Quick Actions menu tells the IDE to add any missing interface members.
- `orderby` and `where` clauses in LINQ queries can be rewritten using the **OrderBy and Where** LINQ methods.
- You can use the => operator to **turn a field into a property** with a get accessor that executes a lambda expression.
- The **?: operator** (called the conditional or ternary operator) lets you create a single expression that executes an if/else condition.
- LINQ methods that take a `Func<T1, T2>` **parameter** can be called with a lambda that takes a T1 parameter and returns a T2 value.
- Use the => operator to create **switch expressions**, which are like `switch` statements that return a value.
- The Enumerable class has static **Range, Empty, and Repeat methods** to help you create enumerable sequences.
- Use **`yield return` statements** to create methods that return enumerable sequences.
- When a method executes a `yield return`, it returns the next value in the sequence. The next time the method is called, it **resumes execution** at the next statement after the last `yield return` that was executed.

Collectioncross



Across

1. Use the `var` keyword to declare an _____ typed variable
7. A collection _____ combines the declaration with items to add
9. What you're trying to make your code when you have lots of tests for weird data and edge cases
11. LINQ method to return the last elements in a sequence
12. A last-in, first-out (LIFO) collection
18. LINQ method to return the first elements in a sequence
19. A method that has multiple constructors with different parameters
20. The type of parameter that tells you that you can use a lambda
21. What you take advantage of when you upcast an entire list
22. What you're using when you call myArray[3]
25. What T gets replaced with when you see <T> in a class or interface definition
32. The keyword you use to create an anonymous object
33. A data type that only allows certain values
34. The kind of collection that can store any type
35. The interface that all sequences implement
36. Another name for the ?: conditional operator

Down

1. If you want to sort a List, its members need to implement this
2. A collection class for storing items in order
3. A collection that stores keys and values
4. What you pass to List.Sort to tell it how to sort its items

Down

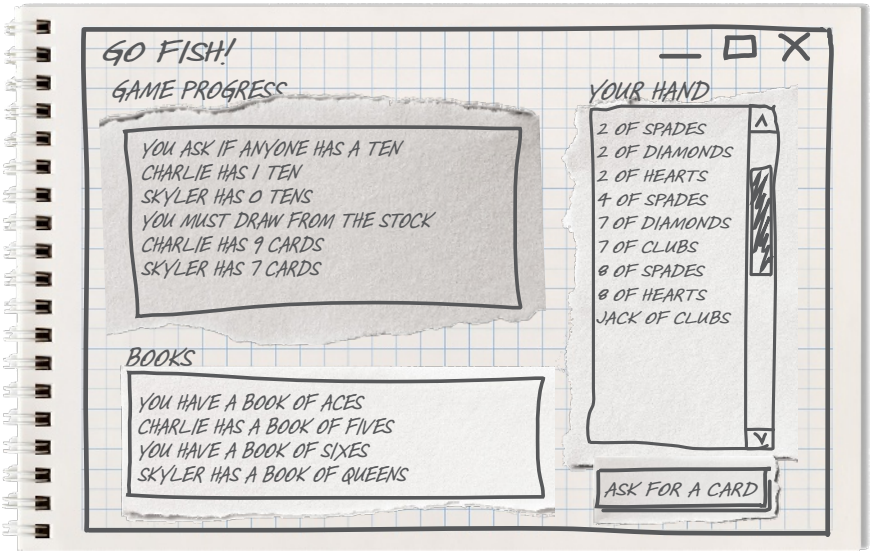
5. What goes in the parentheses: (_____) => expression;
6. You can't use the `var` keyword to declare one of these
8. The access modifier for a class that can't be accessed by another project in a multiproject solution
10. The kind of expression the => operator creates
13. LINQ method to append the elements from one sequence to the end of another
14. Every collection has this method to put a new element into it
15. What you can do with methods in a class that return the type of that class
16. What kind of type you're looking at when the IDE tells you this:

```
'a' is a new string Color, int Height
```

17. An object's namespace followed by a period followed by the class is a fully _____ class name
23. The kind of evaluation that means a LINQ query isn't run until its results are accessed
24. The clause in a LINQ query that sorts the results
26. Type of variable created by the `from` clause in a LINQ query
27. The Enumerable method that returns a sequence with many copies of the same element
28. The clause in a LINQ query that determines which elements in the input to use
29. A LINQ query that merges data from two sequences
30. A first-in, first-out (FIFO) collection
31. The keyword a `switch` statement has that a switch expression doesn't

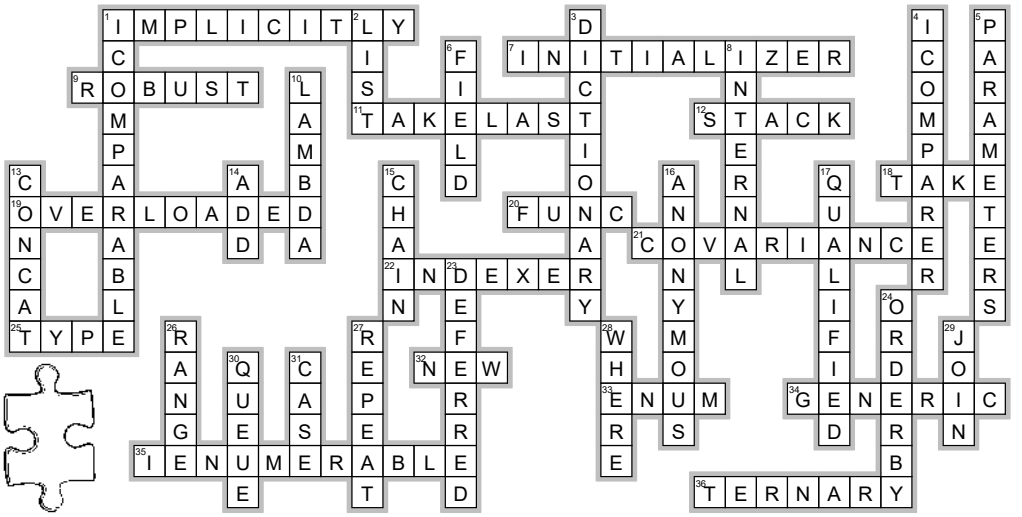
Downloadable exercise: Go Fish

In this next exercise, you'll build a Go Fish card game where you play against computer players. Unit testing will play an important part, because you'll be doing **test-driven development**, a technique where you write your unit tests before you write the code that they test.



Go to the GitHub page for the book and download the project PDF:
https://github.com/head-first-csharp/fifth-edition

Collectioncross solution



10 reading and writing files

Save the last byte for me

Sometimes it pays to be persistent.

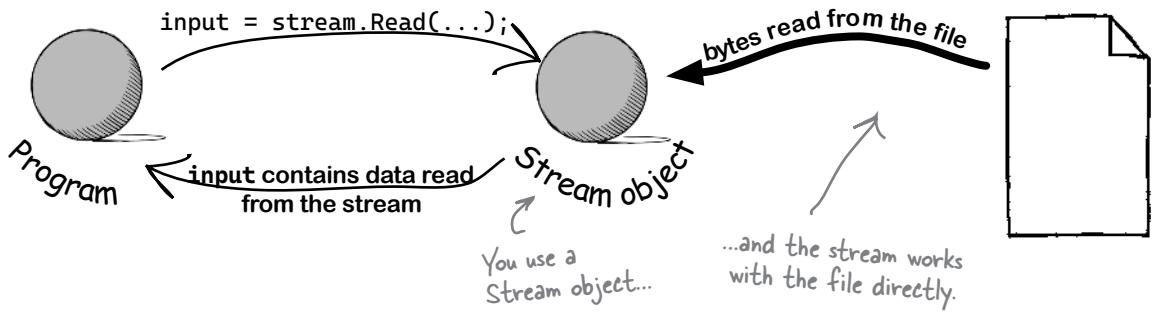
So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**, and then how to **read that information back in** from a file. You'll learn about **streams**, and how to store your objects in files with **serialization**, and get down to the actual bits and bytes of **hexadecimal**, **Unicode**, and **binary data**.

.NET uses streams to read and write data

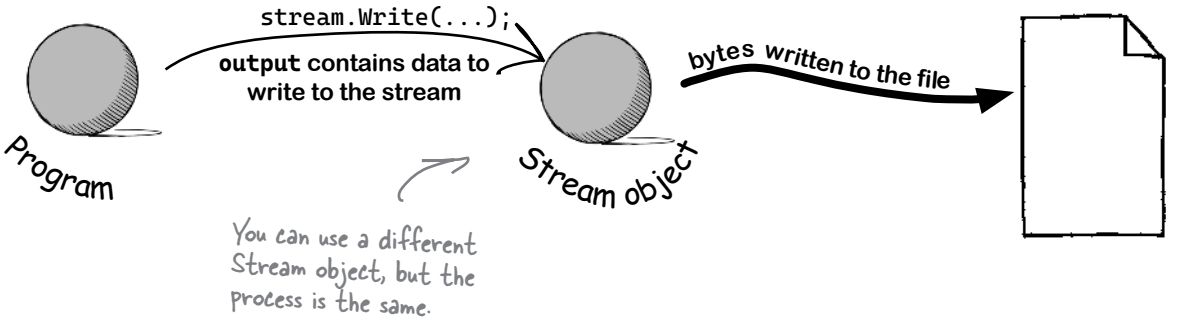
A **stream** is the .NET Framework's way of getting data into and out of your program. Any time your program reads or writes a file, connects to another computer over a network, or generally does anything where it sends or receives bytes, you're using streams. Sometimes you're using streams directly, other times indirectly. Even when you're using classes that don't directly expose streams, under the hood they're almost always using streams.

Whenever you want to read data from a file or write data to a file, you'll use a Stream object.

Let's say you have a simple app that needs to read data from a file. A really basic way to do that is to use a Stream object.

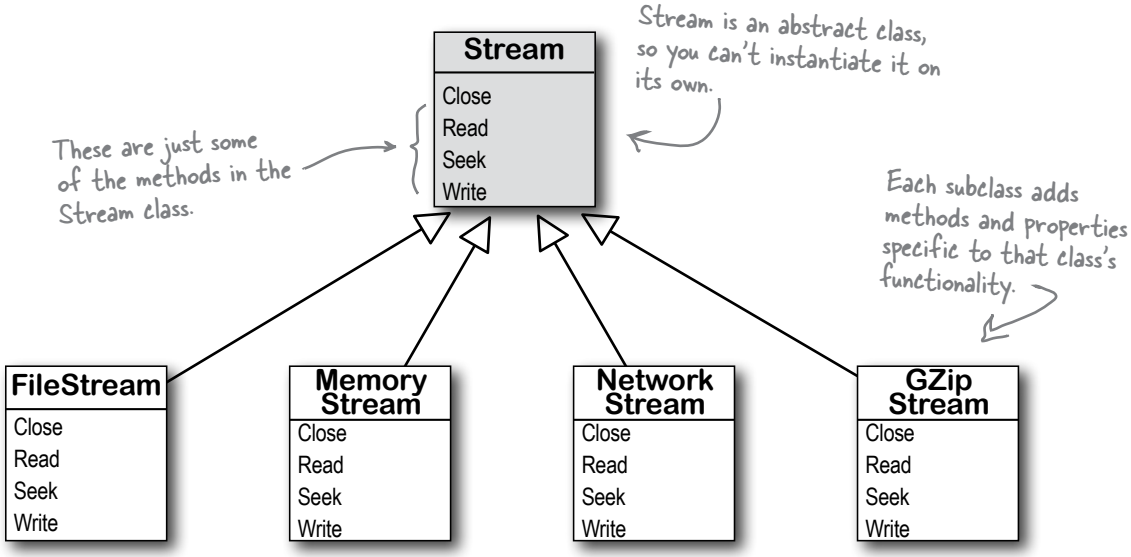


And if your app needs to write data out to the file, it can use another Stream object.



Different streams read and write different things

Every stream is a subclass of the **abstract Stream class**, and there are many subclasses of Stream that do different things. We'll be concentrating on reading and writing regular files, but everything you learn about streams in this chapter can apply to compressed or encrypted files, or network streams that don't use files at all.



A **FileStream** object lets you read from and write to files.

A **MemoryStream** object lets you read from and write data to chunks of memory.

A **NetworkStream** object lets you read data from and write data to other computers or devices on a network.

A **GZipStream** object lets you compress data so that it takes up less space and is easier to download and store.

Things you can do with a stream:

1. **Write to the stream.**
   You can write your data to a stream through a stream's **Write method**.

2. **Read from the stream.**
   You can use the **Read method** to get data from a file, or a network, or memory, or just about anything else using a stream. You can even read data from ***really big*** files, even if they're too big to fit into memory.

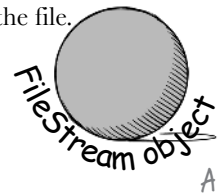
3. **Change your position within the stream.**
   Most streams support a **Seek method** that lets you find a position within the stream so you can read or insert data at a specific place. However, not every Stream class supports Seek—which makes sense, because you can't always backtrack in some sources of streaming data.

Streams let you read and write data. Use the right kind of stream for the data you're working with.

A FileStream reads and writes bytes in a file

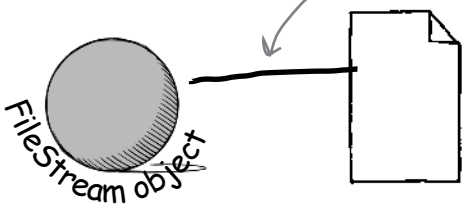
When your program needs to write a few lines of text to a file, there are a lot of things that have to happen:

1. Create a new FileStream object and tell it to write to the file.



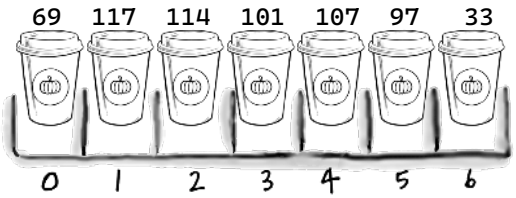
A FileStream can only be attached to one file at a time.

2. The FileStream attaches itself to a file.



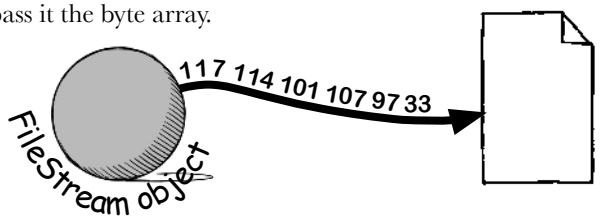
3. Streams write bytes to files, so you'll need to convert the string that you want to write to an array of bytes.

Eureka! →



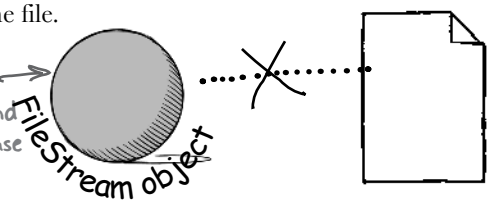
This is called encoding, and we'll talk more about it later on...

4. Call the stream's Write method and pass it the byte array.



5. Close the stream so other programs can access the file.

Forgetting to close a stream is a big deal. The file will be locked and other programs won't be able to use it until you close your stream.



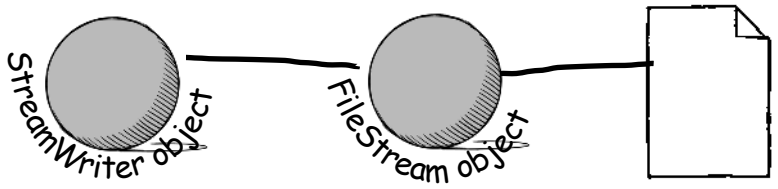
Write text to a file in three simple steps

C# comes with a convenient class called **StreamWriter** that simplifies those things for you. All you have to do is create a new StreamWriter object and give it a filename. It ***automatically*** creates a FileStream and opens the file. Then you can use the StreamWriter's Write and WriteLine methods to write everything to the file you want.

StreamWriter creates and manages a FileStream object for you automatically.

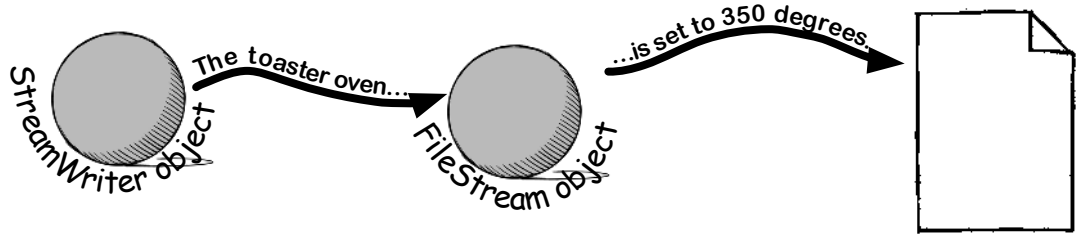
1. **Use the StreamWriter's constructor to open or create a file.**
You can pass a filename to the StreamWriter's constructor. When you do, the writer automatically opens the file. StreamWriter also has an overloaded constructor that lets you specify its *append* mode: passing it `true` tells it to add data to the end of an existing file (or append), while `false` tells the stream to delete the existing file and create a new file with the same name.

```
var writer = new StreamWriter("toaster oven.txt", true);
```



2. **Use the Write and WriteLine methods to write to the file.**
These methods work just like the ones in the Console class: Write writes text, and WriteLine writes text and adds a line break to the end.

```
writer.WriteLine($"The {appliance} is set to {temp} degrees.");
```



3. **Call the Close method to release the file.**
If you leave the stream open and attached to a file, then it'll keep the file locked and no other program will be able to use it. So make sure you always close your files!

```
writer.Close();
```

The Swindler launches another diabolical plan

The citizens of Objectville have long lived in fear of the Swindler, Captain Amazing's archnemesis. Now he's using a StreamWriter to implement another evil plan. Let's take a look at what's going on.

Create a new Console App project and **add this code to the top-level statements**:

```
StreamWriter sw = new StreamWriter("secret_plan.txt");

sw.WriteLine("How I'll defeat Captain Amazing");
sw.WriteLine("Another genius secret plan by The Swindler");
sw.WriteLine("I'll unleash my army of clones upon the citizens of Objectville.");

string location = "the mall";
for (int number = 1; number <= 5; number++)
{
    sw.WriteLine("Clone #{0} attacks {1}", number, location);
    location = (location == "the mall") ? "downtown" : "the mall";
}
sw.Close();
```

This line creates the StreamWriter object and tells it where the file will be.

See if you can figure out what's going on with the location variable and the ?: ternary operator.

It's *really important* that you call Close when you're done with the StreamWriter—that frees up any connections to the file and any other resources that the StreamWriter instance is using. If you don't close the stream, some of the text *might not get written*.

StreamWriter.WriteLine works just like Console.WriteLine

The StreamWriter class has Write and WriteLine methods that work just like the methods in the Console class with the same names: Write writes text, and WriteLine writes text with a line break.

The Write and WriteLine methods classes support {curly brackets} like this:

```
sw.WriteLine("Clone #{0} attacks {1}", number, location);
```

When you include {0} in the text, it's replaced by the first parameter after the string; {1} is replaced by the second, {2} by the third, etc.

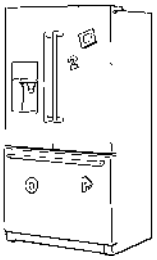
Since you didn't include a full path in the filename, it wrote the output file ***to the same folder as the binary***. Check the bin\Debug\net8.0 folder underneath your Solution folder (if you're using a different version of .NET, the folder under Debug may have a different name).

Here's the output that it writes to *secret_plan.txt*:

Output

```
How I'll defeat Captain Amazing
Another genius secret plan by The Swindler
I'll unleash my army of clones upon the citizens of
Objectville.
Clone #1 attacks the mall
Clone #2 attacks downtown
Clone #3 attacks the mall
Clone #4 attacks downtown
Clone #5 attacks the mall
```

The Swindler is Captain Amazing's archnemesis, a shadowy supervillain bent on the domination of Objectville.



StreamWriter Magnets

Oops! These magnets were nicely arranged on the fridge with the code for the Flobbo class, but someone slammed the door and they all fell off. Here are the top-level statements from the app:

```
Flobbo f = new Flobbo("blue yellow");
StreamWriter sw = f.Snobbo();
f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
```

The magnets make up a class. Can you rearrange them so the top-level statements produce the output below?

We added an extra challenge.

Something weird is going on with the Blobbo method. See how it has two different declarations in the first two magnets? We defined Blobbo as an **overloaded method**—there are two different versions, each with its own parameters, just like the overloaded methods you've used in previous chapters.

```
class Flobbo(string zap)
{
}
```

```
sw.WriteLine(zap);
zap = "green purple";
return false;
```

```
sw.WriteLine(zap);
zap = "red orange";
return true;
```

```
sw.WriteLine(zap);
sw.Close();
return false;
```

```
}
else
{
```

```
if (Already) {
```

```
public bool Blobbo(StreamWriter sw) {
```

```
public StreamWriter Snobbo() {
```

```
return new
   StreamWriter("macaw.txt");
```

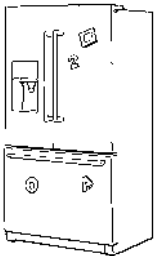
```
public bool Blobbo
    (bool Already, StreamWriter sw)
{
```

```
}
}
}
}
```

Here's the output of the app that gets written to a file called macaw.txt.

Output

```
blue yellow
green purple
red orange
```



StreamWriter Magnets Solution

Your job was to construct the Flobbo class from the magnets to create the desired output.

```
Flobbo f = new Flobbo("blue yellow");
StreamWriter sw = f.Snobbo();
f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
```

Add the code to a console app. The output will be written to `macaw.txt` in the same folder as the binary—a subdirectory under the bin\Debug folder inside the project folder.

Output

```
blue yellow
green purple
red orange
```

```
class Flobbo(string zap)
{
    public StreamWriter Snobbo() {
        return new
            StreamWriter("macaw.txt");
    }

    public bool Blobbo(StreamWriter sw) {
        sw.WriteLine(zap);
        zap = "green purple";
        return false;
    }

    public bool Blobbo
        (bool Already, StreamWriter sw)
    {
        if (Already) {
            sw.WriteLine(zap);
            sw.Close();
            return false;
        }
        else
        {
            sw.WriteLine(zap);
            zap = "red orange";
            return true;
        }
    }
}
```

Make sure you close files when you're done with them. See if you can figure out why this is called after all of the text is written.

Defining overloaded methods

In Chapter 8, you learned how the Random.Shared.Next method is **overloaded**—there are three versions of it, each with a different set of parameters. The Blobbo method is overloaded too—it has two declarations with different parameters:

public bool Blobbo(StreamWriter sw)

and

public bool Blobbo(bool Already, StreamWriter sw)

The two overloaded Blobbo methods are completely separate from each other. They behave differently, just like the different overloaded versions of Random.Next behave differently. If you add those two methods to a class, the IDE will show them as overloaded methods, just like it did with Random.Next.

Just a reminder: we picked intentionally weird variable names and methods in these puzzles because if we used really good names, the puzzles would be too easy! Don't use names like this in your code, OK?

Use a StreamReader to read a file

Let's read the Swindler's secret plans with **StreamReader**, a class that's a lot like StreamWriter—except instead of writing a file, you create a StreamReader and pass it the name of the file to read in its constructor. Its ReadLine method returns a string that contains the next line from the file. You can write a loop that reads lines from it until its EndOfStream field is true—that's when it runs out of lines to read. Add this console app that uses a StreamReader to read one file, and a StreamWriter to write another file:

StreamReader is a class that reads characters from streams, *but it's not a stream itself*. When you pass a filename to its constructor, it creates a stream for you, and closes it when you call its Close method. It also has an overloaded constructor that takes a reference to a Stream.

Your OS might give you a warning about creating files in this folder, depending on your security settings.

This returns the path of the user's Documents folder on Windows, or the user's home directory on macOS. Make sure that you *copy secret_plan.txt into that folder* or you'll get an exception. Check out the SpecialFolder enum to see what other folders you can find.

```
var folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
Console.WriteLine($"Reading and writing files in this folder: {folder}");

StreamReader reader = new(Path.Combine(folder, "secret_plan.txt"));
StreamWriter writer = new(Path.Combine(folder, "emailToCaptainA.txt"));

writer.WriteLine("To: CaptainAmazing@objectville.net");
writer.WriteLine("From: Commissioner@objectville.net");
writer.WriteLine("Subject: Can you save the day...again?");
writer.WriteLine();
writer.WriteLine("We've discovered the Swindler's terrible plan:");

while (!reader.EndOfStream)
{
    var lineFromThePlan = reader.ReadLine();
    writer.WriteLine($"The plan -> {lineFromThePlan}");
}
writer.WriteLine();
writer.WriteLine("Can you help us?");

writer.Close();
reader.Close();
```

We're using the Path.Combine method, which combines a folder and a filename into a path.

The EndOfStream property is true if the reader has finished reading all the data in the file.

This loop reads a line from the reader and writes it out to the writer.

The StreamReader and StreamWriter each created their own streams. Calling their Close methods tells them to close those streams.

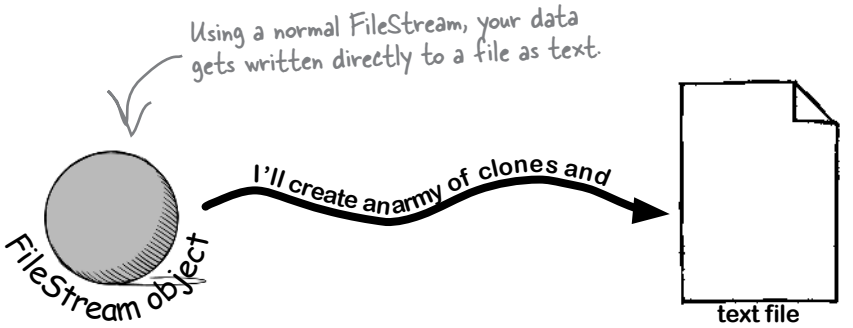
Output

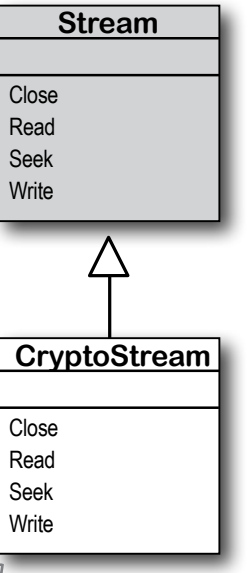
```
To: CaptainAmazing@objectville.net
From: Commissioner@objectville.net
Subject: Can you save the day...again?

We've discovered the Swindler's terrible plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll unleash my army of clones upon the citizens of Objectville.
The plan -> Clone #1 attacks the mall
The plan -> Clone #2 attacks downtown
The plan -> Clone #3 attacks the mall
The plan -> Clone #4 attacks downtown
The plan -> Clone #5 attacks the mall

Can you help us?
```

This output will be in a file called *emailToCaptainA.txt*—check the console output of the app to see the folder location.

Data can go through more than one stream

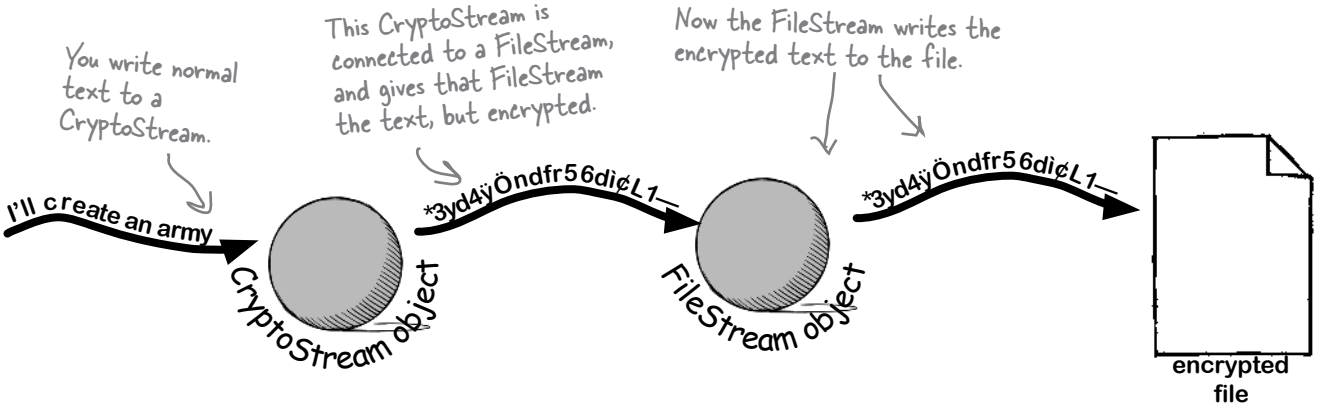
One big advantage to working with streams in .NET is that you can have your data go through more than one stream on its way to its final destination. One of the many types of streams in .NET is the CryptoStream class. This lets you encrypt your data before you do anything else with it. So instead of writing plain text to a regular old text file:





CryptoStream inherits from the abstract Stream class, just like the other stream classes.

the Swindler can **chain streams together** and send the text through a CryptoStream object before writing its output to a FileStream:



You can chain streams. One stream can write to another stream, which writes to another stream...often ending with a network or file stream.

Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the Pineapple, Pizza, and Party classes. You can use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make the program write a file called *order.txt* with the five lines listed in the output box below.

```
const _______ d = "delivery.txt";

______ o = new _____________("order.txt");
var pz = new _______(new _____________(d,
true));
pz._______(Fargo.Flamingo);
for (______ w = 3; w >= 0; w--) {
    var i = new _______(new ____________(d,
false));
    i.Idaho((Fargo)w);
    Party p = new _______(new ____________(d));
    p.HowMuch(o);
}
o._________("That's all folks!");
o._______();
```

```
______ _______
{
    North, South, East, West, Flamingo
}

class Pizza(_____________ _______) {
    public void Idaho(________ f) {
        writer._________(f);
        writer._______();
    }
}

class Party(_____________ ________) {
    public void HowMuch(_____________ q)
{
        q._________(reader._________
());
        reader._______();
    }
}
```

The code writes these lines to the file order.txt.

order.txt

```
West
East
South
North
That's all folks!
```

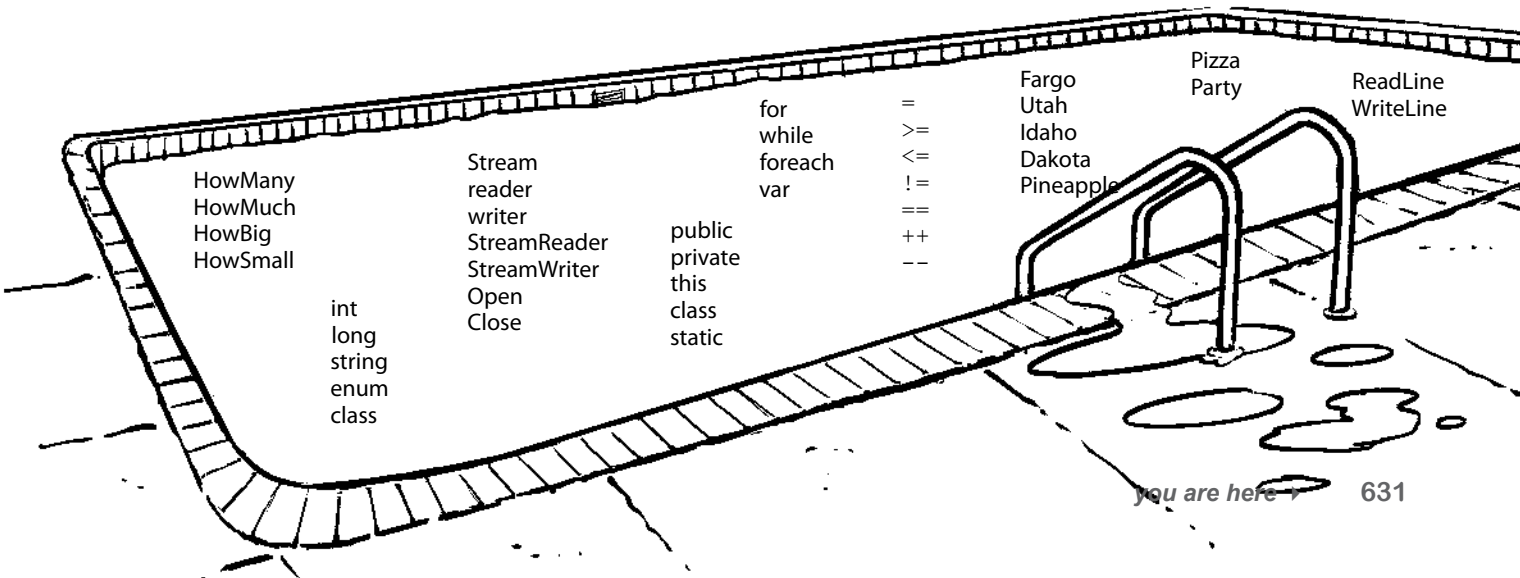
Note: each snippet from the pool can be used more than once!

Here's a bonus question.

MINI **Sharpen your pencil**

What text does the app write to *delivery.txt*?

.....



Pool Puzzle Solution

```
const  string  d = "delivery.txt";

 var  o = new  StreamWriter  ("order.txt");
var pz = new  Pizza  (new  StreamWriter  (d, true));
pz.  Idaho  (Fargo.Flamingo);
for (  int  w = 3; w >= 0; w--) {
    var i = new  Pizza  (new  StreamWriter  (d,
false));
    i.Idaho((Fargo)w);
    Party p = new  Party  (new  StreamReader  (d));
    p.HowMuch(o);
}
o.  WriteLine  ("That's all folks!");
o.  Close  ();


 enum  Fargo
{
    North, South, East, West, Flamingo
}

class Pizza(  StreamWriter  writer  ) {
    public void Idaho(  Fargo  f) {
        writer.  WriteLine  (f);
        writer.  Close  ();
    }
}

class Party(  StreamReader  reader  ) {
    public void HowMuch(  StreamWriter  q) {
        q.  WriteLine  (reader.  ReadLine  ());
        reader.  Close  ();
    }
}
```

The top-level statements create a StreamWriter and pass it to the Party class. Then they loop through the Fargo members, passing each of them to the Pizza.Idaho method to print.

This enum is used to print a lot of the output. We learned in Chapter 8 that an enum's ToString method returns the equivalent string, so Fargo.North.ToString() returns the string "North."

The Pizza class take a StreamWriter as an argument for its primary constructor. Its Idaho method uses it to write Fargo enums to the file using their ToString methods, which WriteLine calls automatically.

The Party class has a primary constructor with a StreamReader argument, and its HowMuch method reads a line from that StreamReader and writes it to a StreamWriter.

Here's the output the app writes to the order.txt file.

order.txt

```
West
East
South
North
That's all folks!
```

MINI **Sharpen your pencil** Solution

What text does the app write to *delivery.txt*?

North

there are no Dumb Questions

Q: Can you explain what you were doing with {0} and {1} when you called the StreamWriter Write and WriteLine methods?

A: When you're printing strings to a file, you'll often find yourself in the position of having to print the contents of a bunch of variables. For example, you might have to write something like this:

```
writer.WriteLine("My name is " + name +
    "and my age is " + age);
```

It gets really tedious and somewhat error-prone to have to keep using + to combine strings. It's easier to use **composite formatting**, where you use a **format string with placeholders** like {0}, {1}, {2}, etc., and follow it with variables to replace the placeholders:

```
writer.WriteLine(
 "My name is {0} and my age is {1}", name, age);
```

You're probably thinking, isn't that really similar to string interpolation? And you're right—it is! In some cases string interpolation may be easier to read, and in other cases using a format string is cleaner. Just like string interpolation, **format strings support formatting**. For example, {1:0.00} means format the second argument as a number with two decimal places, while {3:c} says to format the fourth argument in the local currency.

Oh, and one more thing—format strings work with Console.Write and Console.WriteLine too!

Q: What was that Path.DirectorySeparatorChar field that you used in the console app that used StringReader?

A: We wrote that code to work on both Window and macOS, so we took advantage of some of .NET's tools to help with that. Windows uses backslash characters as a path separator (C:\Windows), while macOS uses a forward slash (/Users).

Path.DirectorySeparatorChar is a read-only field that's set to the correct path separator character for the operating system: a \ on Windows and / on macOS and Linux.

We also used the Environment.GetFolderPath method, which returns the path of one of the special folders for the current user: the user's Documents folder (Windows) or home directory (macOS/Linux).

Q: Near the beginning of the chapter, you talked about converting a string to a byte array. How would that even work?

A: You've probably heard many times that files on disk are represented as bits and bytes. What that means is that when you write a file to a disk, the operating system treats it as one long sequence of bytes. The StreamReader and StreamWriter are converting from bytes to characters for you—that's called *encoding* and *decoding*. Remember from Chapter 4 how a byte variable can store any number between 0 and 255? Every file on your hard drive is one long sequence of numbers between 0 and 255. It's up to the programs that read and write those files to interpret those bytes as meaningful data. When you open a file in Notepad, it converts each individual byte to a character—for example, `E` is 69 and `a` is 97 (but this depends on the encoding…you'll learn more about encodings in just a minute). When you type text into Notepad and save it, Notepad converts each character back into a byte and saves it to disk. If you want to write a string to a stream, you'll need to do the same.

Q: If I'm just using a StreamWriter to write to a file, why do I really care if it's creating a FileStream for me?

A: If you're only reading or writing lines to or from a text file in order, then all you need are StreamReader and StreamWriter. As soon as you need to do anything more complex than that, you'll need to start working with other streams. If you ever need to write data like numbers, arrays, collections, or objects to a file, a StreamWriter just won't do. We'll go into a lot more detail about how that will work in just a minute.

Q: Why do I need to worry about closing streams after I'm done with them?

A: Have you ever had an app tell you that it couldn't open a file because it was "busy"? When one program uses a file, your operating system locks it and prevents other programs from using it. The apps you write are no exception—Windows or macOS will do that for your apps when they open files too. If you don't call the Close method, then it's possible for your program to keep a file locked until it ends.

Console and StreamWriter can use composite formatting, which replaces placeholders with values of parameters passed to Write or WriteLine.

Use the static File and Directory classes to work with files and directories

Like StreamWriter, the File class creates streams that let you work with files behind the scenes. You can use its methods to do most common actions without having to create the FileStreams first. The Directory class lets you work with whole directories full of files.

Things you can do with the static File class:

1. **Find out if the file exists.**
   You can check to see if a file exists using the File.Exists method. It'll return true if it does, and false if it doesn't.

2. **Read from and write to the file.**
   The File.OpenRead method lets you open a file to read data from. File.Create creates a new file and writes to it, and File.OpenWrite opens an existing file for writing.

3. **Append text to the file.**
   The File.AppendAllText method lets you append text to an already created file. It even creates the file if it's not there when the method runs.

4. **Get information about the file.**
   The File.GetLastAccessTime and File.GetLastWriteTime methods return the date and time when the file was last accessed and modified.

FileInfo works just like File

If you're going to be doing a lot of work with a file, you might want to create an instance of the FileInfo class instead of using the File class's static methods.

The FileInfo class does just about everything the File class does, except you have to instantiate it to use it. You can create a new instance of FileInfo and access its Exists method or its OpenRead method in just the same way.

The big difference is that the File class is faster for a small number of actions, and FileInfo is better suited for big jobs.

File is a static class, so it's just a set of methods that let you work with files. FileInfo is an object that you instantiate, and its methods are the same as the ones you see on File.

Things you can do with the static Directory class:

1. **Create a new directory.**
   Create a directory using the Directory.CreateDirectory method. All you have to do is supply the path; this method does the rest.

2. **Get a list of the files in a directory.**
   You can create an array of files in a directory using the Directory.GetFiles method; just tell the method which directory you want to know about, and it will do the rest.

3. **Delete a directory.**
   Need to delete a directory? Call the Directory.Delete method.

Sharpen your pencil

.NET has classes with a bunch of static methods for working with files and folders, and their method names are intuitive. The File class gives you methods to work with files, and the Directory class lets you work with directories. Write down what you think each of these lines of code does, then answer the two additional questions at the end.

Code	What the code does

`if (!Directory.Exists(@"C:\SYP")) {`
`  Directory.CreateDirectory(@"C:\SYP");`
`}`	

`if (Directory.Exists(@"C:\SYP\Bonk")) {`
`  Directory.Delete(@"C:\SYP\Bonk");`
`}`	

`if (File.Exists(@"C:\SYP\Bonk\Whammo.txt"))` `  File.Delete(@"C:\SYP\Bonk\Whammo.txt");`	
`Directory.SetCreationTime(@"C:\SYP\Bonk",` `  new DateTime(1996, 09, 23));`	
`string[] files = Directory.GetFiles(@"C:\SYP\",` `  "*.log", SearchOption.AllDirectories);`	

`File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt",`
`  @"This is the first line`
`and this is the second line`
`and this is the last line");`	

`File.Encrypt(@"C:\SYP\Bonk\weirdo.txt");` See if you can guess what this method does—you haven't seen it yet.	
`File.Copy(@"C:\SYP\Bonk\weirdo.txt",` `  @"C:\SYP\copy.txt");`	
`DateTime myTime =` `  Directory.GetCreationTime(@"C:\SYP\Bonk");`	
`File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);`	
`File.Delete(@"C:\SYP\Bonk\weirdo.txt");`	

Why did we put @ in front of each of the strings that we passed as arguments to the methods above?

.....

.....

The filenames above all start with C:\ to work on Windows. What happens if you run that code on macOS or Linux?

.....

.....

Sharpen your pencil Solution

Code	What the code does

`if (!Directory.Exists(@"C:\SYP")) {`
`  Directory.CreateDirectory(@"C:\SYP");`
`}`
	Check if the C:\SYP folder exists. If it doesn't, create it.

`if (Directory.Exists(@"C:\SYP\Bonk")) {`
`  Directory.Delete(@"C:\SYP\Bonk");`
`}`
	Check if the C:\SYP\Bonk folder exists. If it does, delete it.

`if (File.Exists(@"C:\SYP\Bonk\Whammo.txt"))` `  File.Delete(@"C:\SYP\Bonk\Whammo.txt");`	Check if the file C:\SYP\Bonk\Whammo.txt exists. If it does, delete it.
`Directory.SetCreationTime(@"C:\SYP\Bonk",` `  new DateTime(1996, 09, 23));`	Set the creation time for the C:\SYP\Bonk folder to September 23, 1996.
`string[] files = Directory.GetFiles(@"C:\SYP\",` `  "*.log", SearchOption.AllDirectories);`	Get a list of all files in C:\SYP that match the *.log pattern, including all matching files in any subdirectory.

`File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt",`
`  @"This is the first line`
`and this is the second line`
`and this is the last line");`
	Create a file called "weirdo.txt" (if it doesn't already exist) in the C:\SYP\Bonk folder and write three lines of text to it.

`File.Encrypt(@"C:\SYP\Bonk\weirdo.txt");` This is an alternative to using a CryptoStream.	Take advantage of built-in Windows encryption to encrypt the file "weirdo.txt" using the logged-in account's credentials.
`File.Copy(@"C:\SYP\Bonk\weirdo.txt",` `  @"C:\SYP\copy.txt");`	Copy the C:\SYP\Bonk\weirdo.txt file to C:\SYP\Copy.txt.
`DateTime myTime =` `  Directory.GetCreationTime(@"C:\SYP\Bonk");`	Declare the myTime variable and set it equal to the creation time of the C:\SYP\Bonk folder.
`File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);`	Alter the last write time of the copy.txt file in C:\SYP\ so it's equal to whatever time is stored in the myTime variable.
`File.Delete(@"C:\SYP\Bonk\weirdo.txt");`	Delete the C:\SYP\Bonk\weirdo.txt file.

Why did we put @ in front of each of the strings that we passed as arguments to the methods above?

The @ keeps the backslashes in the string from being interpreted as escape sequences.

The filenames above all start with C:\ to work on Windows. What happens if you run that code on macOS or Linux?

It will create a filename that starts with "C:\" and put it in the same folder as the binary.

IDisposable makes sure objects are closed properly

A lot of .NET classes implement a particularly useful interface called **IDisposable**. It has only one member: a method called Dispose. Whenever a class implements IDisposable, it's telling you that there are important things that it needs to do in order to shut itself down, usually because the object has ***allocated resources*** that it won't give back until you tell it to. The Dispose method is how you tell the object to release those resources.

Use the IDE to explore IDisposable

Create a new class in any project that implements IDisposable. Use the IDE's Quick Actions and Refactorings (Ctrl+. or ⌘+.) to implement the interface. The IDE will add a single method called Dispose.

Now **hover your mouse cursor over IDisposable** to see the IntelliSense pop-up with the description of the interface. Then **hover over Dispose()** to see the description of the method. Here's what you should see:

A lot of objects allocate important resources, like memory, files, and other objects. That means they take them over, and don't give them back until you tell the object that you're done with those resources.

```
class MyClass : IDisposable
```

interface System.IDisposable
Provides a mechanism for releasing unmanaged resources.

```
{
    void IDisposable.Dispose()
    {
```

void MyClass.Dispose()
Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.

```
        throw new NotImplementedException();
    }
}
```

Any class that implements IDisposable must immediately release any resources that it took over as soon as you call its Dispose method. It's almost always the last thing you do before you're done with the object.

al-lo-cate, verb.
to distribute resources or duties for a particular purpose. *The programming team was irritated at their project manager because he* ***allocated*** *all of the conference rooms for a useless management seminar.*

Any class that implements the IDisposable interface can do the job of disposing of any unmanaged resources that it allocated.

Avoid filesystem errors with using statements

Throughout this chapter, we've been stressing that you need to **close your streams**. That's because some of the most common bugs that programmers run across when they deal with files are caused when streams aren't closed properly. Luckily, C# gives you a great tool to make sure that never happens to you: IDisposable and the Dispose method. When you **wrap your stream code in a `using` statement**, it automatically closes your streams for you. All you need to do is **declare your stream reference** with a `using` statement, followed by a block of code (inside curly brackets) that uses that reference. When you do that, C# **automatically calls the Dispose method** as soon as it finishes running the block of code.

These "using" statements are different from the ones at the top of your code.

A using statement is always followed by an object declaration...

...and then a block of code within curly braces.

```
using (var sw = new StreamWriter("secret_plan.txt")) {
   sw.WriteLine("How I'll defeat Captain Amazing");
   sw.WriteLine("Another genius secret plan");
   sw.WriteLine("by The Swindler");
}
```

After the last statement in the using block executes, it calls the Dispose method of the object being used.

In this case, the object being used is pointed to by sw—which was declared in the using statement—so the Dispose method of the Stream class is called automatically, which closes the stream.

This `using` statement declares a variable `sw` that references a new StreamWriter and is followed by a block of code. After all of the statements in the block are executed, the `using` block will *automatically call sw.Dispose*.

Use multiple using statements for multiple objects

You can pile `using` statements on top of each other—you don't need extra sets of curly brackets or indents:

```
using (var reader = new StreamReader("secret_plan.txt"))
using (var writer = new StreamWriter("email.txt"))
{
    // statements that use reader and writer
}
```

Every stream has a Dispose method that closes the stream. When you declare your stream in a `using` statement, it will *always get closed*! And that's important, because some streams *don't write all of their data until they're closed*.

When you declare an object in a using block, that object's Dispose method is called automatically.

Use a MemoryStream to stream data to memory

We've been using streams to read and write files. What if you want to read data from a file and then, well, do something with it? You can use a **MemoryStream**, which keeps track of all data streamed to it by storing it in memory. For example, you can create a new MemoryStream and pass it as an argument to a StreamWriter constructor, and then any data you write with the StreamWriter will be sent to that MemoryStream. You can retrieve that data using the **MemoryStream.ToArray method**, which returns all of the data that's been streamed to it in a byte array.

Use Encoding.UTF8.GetString to convert byte arrays to strings

One of the most common things that you'll do with byte arrays is convert them to strings. For example, if you have a byte array called bytes, here's one way to convert it to a string:

```
var converted = Encoding.UTF8.GetString(bytes);
```

We mentioned "encoding" earlier. What do you think that means?

Sleuth it Out

Here's a small console app that uses composite formatting to write a number to a MemoryStream, and then convert it to a byte array and then to a string. Just one problem…***it doesn't work!***

Create a new console app and add this code to it. Can you sleuth out the problem and fix it?

```
using System.Text;

using (var ms = new MemoryStream())
using (var sw = new StreamWriter(ms))
{
    sw.WriteLine("The value is {0:0.00}", 123.45678);
    Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
}
```

The second using statement uses the variable created in the first one.

The MemoryStream.ToArray method returns all of the streamed data as a byte array. The GetString method converts that byte array to a string.

When you run this app, it's supposed to write a line of text to the console, but it doesn't write anything at all. Before we show you what went wrong, see if you can *sleuth it out* on your own.

Here's a hint: can you figure out when the streams are closed?

there are no Dumb Questions

Q: Remind me why you put an @ in front of the strings that contained filenames in that "Sharpen your pencil" exercise?

A: Because if we didn't, the \S in "C:\SYP" would be interpreted as an invalid escape sequence and throw an exception. When you add a string literal to your program, the compiler converts escape sequences like `\n` and `\r` to special characters. Windows filenames have backslash characters in them, but C# strings normally use backslashes to start escape sequences. If you put @ in front of a string, it tells C# not to interpret escape sequences. It also tells C# to include line breaks in your string, so you can hit Enter halfway through the string and it'll include that as a line break in the output.

Q: And what exactly are escape sequences?

A: An escape sequence is a way to include special characters in your strings. For example, `\n` is a line feed, `\t` is a tab, and `\r` is a return character, or half of a Windows return (in Windows text files, lines have to end with `\r\n`; for macOS and Linux, lines just end in `\n`). If you need to include a quotation mark inside a string, you can use `\"` and it won't end the string. If you want to use an actual backslash in your string and not have C# interpret it as the beginning of an escape sequence, just do a double backslash: `\\`.

Sleuth it Out

Sherlock Holmes once said, "Data! Data! Data! I can't make bricks without clay." Let's start at the scene of the crime: our code that's not working. We'll *scour it for all of the data that we can find* by digging up clues.

How many of these clues did you spot?

- We instantiate a StreamWriter that feeds data into a new MemoryStream.
- The StreamWriter writes a line of text to the MemoryStream.
- The contents of the MemoryStream are copied to an array and converted to a string.
- This all happens inside a `using` block, so the streams are definitely closed.

If you spotted all of those clues, then congratulations—you've been sharpening your code detective skills! But like in every great mystery, there's always one last clue, some fact we learned earlier that proves to be the key to unraveling the whole crime and finding the culprit.

We used a `using` block, so we know that the streams definitely get closed. ***But* when *are they closed?*** And that leads us to the keystone of this mystery, the all-important clue that we learned just moments before the crime: ***some streams don't write all of their data* until they're closed**.

The StreamWriter and MemoryStream are declared in the same `using` block, so both Dispose methods are called *after the last line in the block is executed*. What does that mean? It means the MemoryStream.ToArray method is called ***before*** *the StreamWriter is closed*.

So we can fix the problem by adding a ***nested*** `using` block to *first* close the StreamWriter and *then* call ToArray:

```
using System.Text;

using (var ms = new MemoryStream())
{
    using (var sw = new StreamWriter(ms))
    {
        sw.WriteLine("The value is {0:0.00}", 123.45678);
    }
    Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
}
```

Your app needs this using directive because the Encoding class is in the System.Text namespace.

Stream objects often have data in memory that's *buffered*, or waiting to be written. When the stream empties all of that data, it's called *flushing*. If you need to flush the buffered data without closing the stream, you can *also call its Flush method*.

Exercise

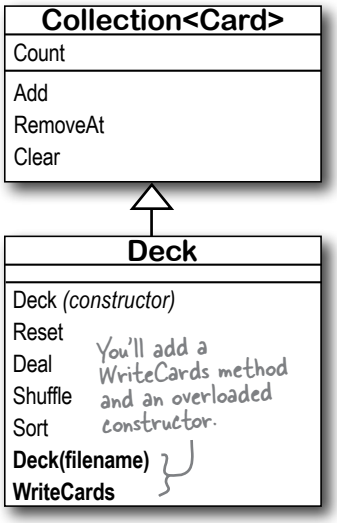
In Chapter 8, you created a Deck class that had a sequence of Card objects, with methods to reset it to an ordered 52-card deck, shuffle the cards to randomize their order, and sort the cards to put them back in order. Now you'll add a method to write the cards to a file, and a constructor that lets you initialize a new deck by reading cards from a file.

Start by reviewing the Deck and Card classes that you wrote in Chapter 8

You created your Deck class by extending a generic collection of Card objects. This allowed you to use some useful members that Deck inherited from Collection<Card>:

- The Count property returns the number of cards in the deck.
- The Add method adds a card to the top of the deck.
- The RemoveAt method removes a card at a specific index from the deck.
- The Clear method removes all cards from the deck.

That gave you a solid starting point to add a Reset method that clears the deck and then adds 52 cards in order (Ace through King in each suit), a Deal method to remove the card from the top of the deck and return it, a Shuffle method to randomize the order of the cards, and a Sort method to put them back in order.



Add a method to write all of the cards in the deck to a file

Your Card class has a Name property that returns a string like "Three of Clubs" or "Ace of Hearts." Add a method called WriteCards that takes a string with a filename as a parameter and writes the name of each card to a line in that file—so if you reset the deck and then call WriteCards, it will write 52 lines to the file, one for each card.

Add an overloaded Deck constructor that reads a deck of cards from a file

Add a second constructor to the Deck class. Here's what it should do:

The String.Split method lets you specify an array of separator characters (in this case, a space), uses them to split the string into parts, and returns an array with each part.

```
public Deck(string filename)
{
  // Create a new StreamReader to read the file.
  // For each line in the file, do the following four things:
  // Use the String.Split method: var cardParts = nextCard.Split(new char[] { ' ' });
  // Use a switch expression to get each card's suit: var suit = cardParts[2] switch {
  // Use a switch expression to get each card's value: var value = cardParts[0] switch {
  // Add the card to the deck.
}
```

In Chapter 9, you learned that switch expressions must be exhaustive, so add a default case that **throws a new InvalidDataException** if it encounters a suit or value that it doesn't recognize—this will make sure each card is valid.

Here are top-level statements you can use to test your class. They create a deck with 10 random cards, write it to a file, and then read that file into a second deck and write each of its cards to the console.

```
var filename = "deckofcards.txt";
Deck deck = new Deck();
deck.Shuffle();
for (int i = deck.Count - 1; i > 9; i--)
    deck.RemoveAt(i);

deck.WriteCards(filename);

Deck cardsToRead = new Deck(filename);
foreach (var card in cardsToRead)
    Console.WriteLine(card.Name);
```

Exercise Solution

Here are the two methods that you added to the Deck class. The WriteCards method uses a StreamWriter to write each card to a file, and the overloaded Deck constructor uses a StreamReader to read each card from a file.

```
public void WriteCards(string filename)
{
   using (var writer = new StreamWriter(filename))
   {
      for (int i = 0; i < Count; i++)
      {
         writer.WriteLine(this[i].Name);
      }
   }
}

public Deck(string filename)
{
   using (var reader = new StreamReader(filename))
   {
      while (!reader.EndOfStream)
      {
         var nextCard = reader.ReadLine();
         var cardParts = nextCard.Split(new char[] { ' ' });
         var value = cardParts[0] switch
         {
            "Ace" => Values.Ace,
            "Two" => Values.Two,
            "Three" => Values.Three,
            "Four" => Values.Four,
            "Five" => Values.Five,
            "Six" => Values.Six,
            "Seven" => Values.Seven,
            "Eight" => Values.Eight,
            "Nine" => Values.Nine,
            "Ten" => Values.Ten,
            "Jack" => Values.Jack,
            "Queen" => Values.Queen,
            "King" => Values.King,
            _ => throw new InvalidDataException($"Unrecognized card value: {cardParts[0]}")
         };
         var suit = cardParts[2] switch
         {
            "Spades" => Suits.Spades,
            "Clubs" => Suits.Clubs,
            "Hearts" => Suits.Hearts,
            "Diamonds" => Suits.Diamonds,
            _ => throw new InvalidDataException($"Unrecognized card suit: {cardParts[2]}"),
         };
         Add(new Card(value, suit));
      }
   }
}
```

This line tells C# to split the nextCard string using a space as a separator character. That splits the string "Six of Diamonds" into the array {"Six", "of", "Diamonds"}.

This switch expression checks the first word in the line to see if it matches a value. If it does, the right Value enum is assigned to the "value" variable.

The default cases in the switch expressions throw an exception if the file contains an invalid card.

We do the same thing for the third word in the line, except we convert this one to a Suit enum.

Have you noticed that the exercises are getting a little more challenging? We're asking you to figure more things out on your own—like how to use the String.Split method. Figuring things out on your own is one of the most important developer skills you can work on.

Bullet Points

- Whenever you want to read data from a file or write data to a file, you'll use a **Stream** object. Stream is an abstract class, with subclasses that do different things.
- A **FileStream** lets you read from and write to files. A **MemoryStream** reads or writes data in memory.
- You can write your data to a stream through a stream's **Write method**, and read data using its **Read method**.
- A **StreamWriter** is a quick way to write data to a file. StreamWriter creates and manages a FileStream object for you automatically.
- A **StreamReader** reads characters from streams, but it's not a stream itself. It creates a stream for you, reads from it, and closes it when you call its Close method.
- The Write and WriteLine methods of StreamWriter and Console use **composite formatting**, which takes a format string with placeholders like {0}, {1}, {2} that support formatting like {1:0.00} and {3:c}.
- **Path.DirectorySeparatorChar** is a read-only field that's set to the path separator character for the operating system: a "\" on Windows and "/" on macOS and Linux.
- The **Environment.GetFolderPath method** returns the path of one of the special folders for the current user, such as the user's Documents folder on Windows or home directory on macOS.
- The **File class** has static methods including Exists (which checks if a file exists), OpenRead and OpenWrite (to get streams to read from or write to the file), and AppendAllText (to write text to a file in one statement).
- The **Directory class** has static methods including CreateDirectory (to create folders), GetFiles (to get the list of files), and Delete (to remove a folder).
- The **FileInfo class** is similar to the File class, except instead of using static methods, it's instantiated.
- Remember to **always close a stream** after you're done with it. Some streams don't write all of their data until they're closed or their **Flush** methods are called.
- The **IDisposable interface** makes sure objects are closed properly. It includes one member, the Dispose method, which provides a mechanism for releasing unmanaged resources.
- Use a **`using` statement** to instantiate a class that implements IDisposable. The `using` statement is followed by a block of code; the object instantiated in the `using` statement is disposed of at the end of the block.
- Use **multiple `using` statements** in a row to declare objects that are disposed of at the end of the same block.

Windows and macOS have different line endings

If you're running Windows, open Notepad. If you're running macOS, open TextEdit. Create a file with two lines—the first line has the characters L1 and the second has the characters L2.

If you used Windows, it will contain these six bytes: 76 49 13 10 76 50.

If you used macOS, it will contain these five bytes: 76 49 10 76 50.

Can you spot the difference? You can see that the first and second lines are encoded with the same bytes: L is 76, 1 is 49, and 2 is 50. The line break is encoded differently: on Windows it's encoded with two bytes, 13 and 10, while on macOS it's encoded with one byte, 10. This is the difference between Windows-style and Unix-style line endings (macOS is a flavor of Unix). If you need to write code that runs on different operating systems and writes files with line endings, you can use the static **Environment.NewLine** property, which returns "\r\n" on Windows and "\r" on macOS or Linux.

All that code just to read one simple card? That's way too much work! What if my object has a ***whole bunch of properties****? Are you telling me I need to write a switch statement for each of them?*

There's an easier way to store your objects in files. It's called serialization.

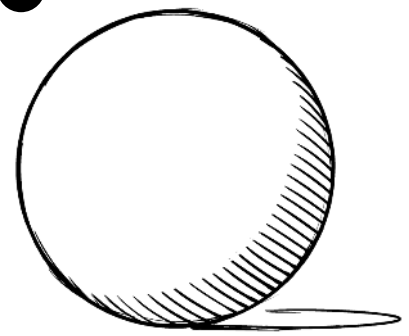
Serialization means writing an object's entire state to a file or string. **Deserialization** means reading the object's state back from that file or string. So instead of painstakingly writing out each field and value to a file line by line, you can save your object the easy way by serializing it out to a stream. ***Serializing*** an object is like **flattening it out** so you can slip it into a file. On the other end, ***deserializing*** an object is like taking it out of the file and **inflating** it again.

OK, just to come clean here: there's also a method called Enum.Parse that will convert the string "Spades" to the enum value Suits.Spades. It even has a companion, Enum.TryParse, which works just like the int.TryParse method you've used throughout this book. But serialization still makes a lot more sense here. You'll find out more about that shortly...

What happens to an object when it's serialized?

It seems like something mysterious has to happen to an object in order to copy it off of the heap and put it into a file, but it's actually pretty straightforward.

1 Object on the heap

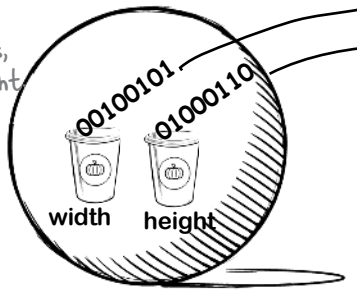


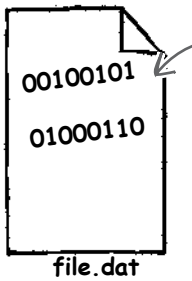
When you create an instance of an object, it has a **state**. Everything that an object "knows" is what makes one instance of a class different from another instance of the same class.

2 Object serialized

When C# serializes an object, it ***saves the complete state of the object***, so that an identical instance (object) can be brought back to life on the heap later.

This object has two byte fields, width and height

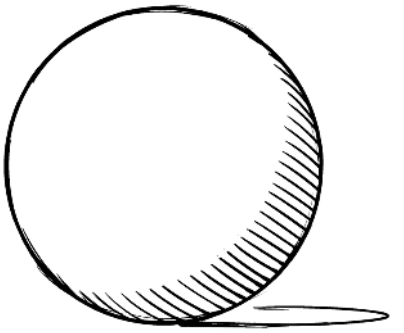




The field values for width and height are saved to the file.dat file, along with a little more info that code needs to restore the object later (like the types of the object and each of its fields).

3 And later on...

Later—maybe days later, and in a different program—you can go back to the file and **deserialize** it. That pulls the original class back out of the file and restores it ***exactly as it was***, with all of its fields and values intact.



Object on the heap again

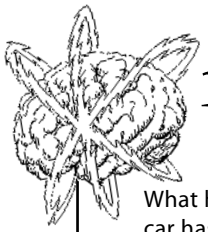
But what exactly IS an object's state? What needs to be saved?

We already know that **an object stores its state in its fields and properties**. So when an object is serialized, each of those values needs to be saved to the file.

Serialization starts to get interesting when you have more complicated objects. Chars, ints, doubles, and other value types have bytes that can just be written out to a file as is. What if an object has an instance variable that's an object reference? What about an object that has five instance variables that are object references? What if those object instance variables themselves have instance variables?

Think about it for a minute. What part of an object is potentially unique? Think about what needs to be restored in order to get an object that's identical to the one that was saved. Somehow everything on the heap has to be written to the file.

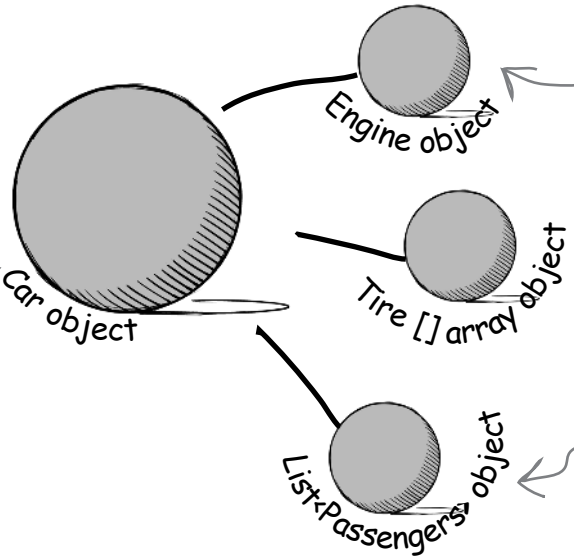
Give your brain a workout and really take a few minutes and really think about this. Can you figure out what the Car object's state would be?



Brain Power

What has to happen for this Car object to be saved so that it can be restored to its original state? Let's say the car has three passengers and a three-liter engine and all-weather radial tires…aren't those things all part of the Car object's state? What should happen to them?

The Car object has references to an Engine object, an array of Tire objects, and a List<> of Passenger objects. Those are part of its state too—what happens to them?



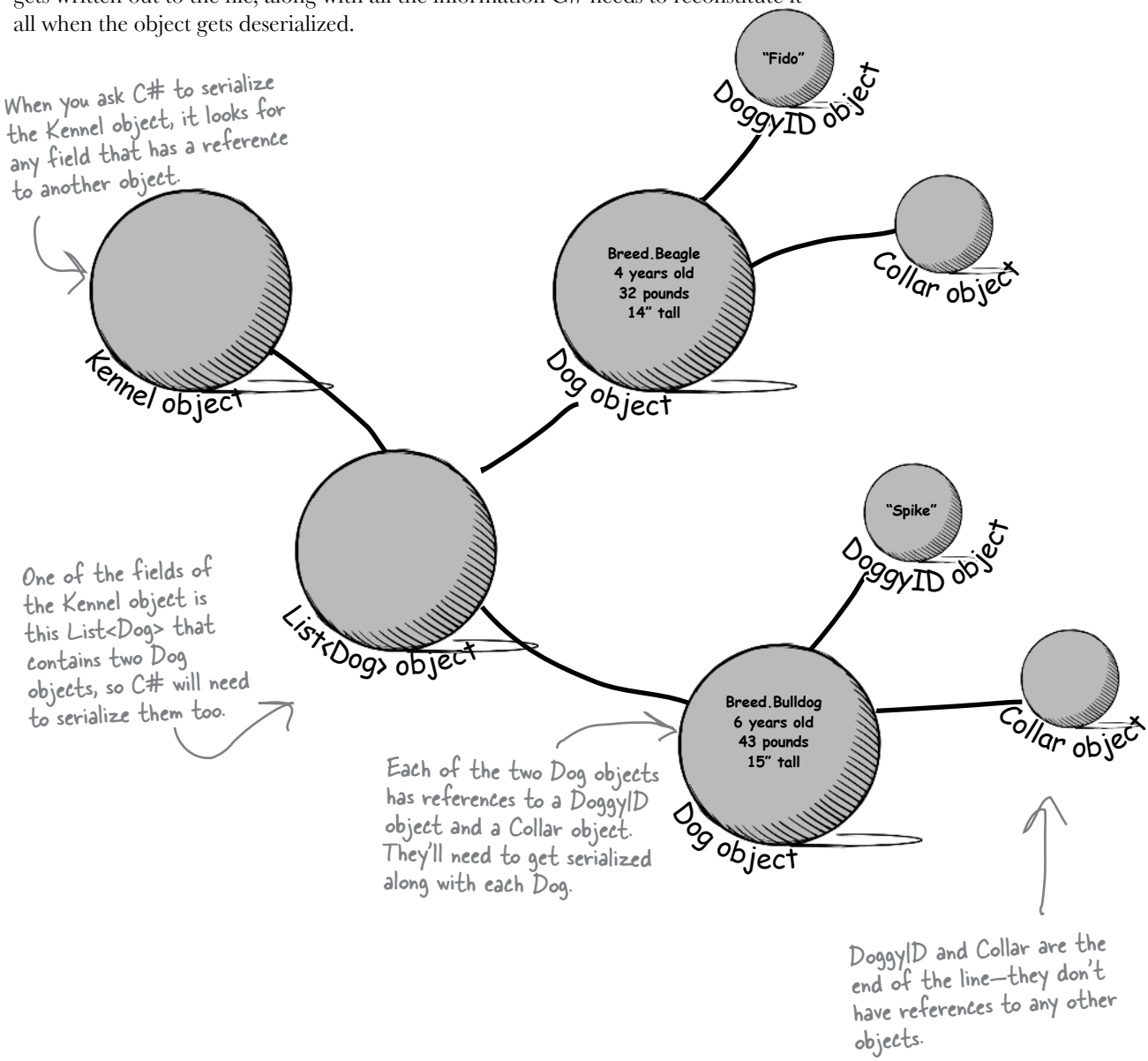
The Engine object is private. Should it be saved too?

Each of the Passenger objects has its own references to other objects. Do those need to be saved too?

When an object is serialized, all of the objects it refers to get serialized too...

A group of objects connected to each other by references is sometimes referred to as a *graph*.

...and all of the objects *they* refer to, and all of the objects *those other objects* refer to, and so on and so on. Don't worry—it may sound complicated, but it all happens automatically. C# starts with the object you want to serialize and looks through its fields for other objects. Then it does the same for each of them. Every single object gets written out to the file, along with all the information C# needs to reconstitute it all when the object gets deserialized.



Use JsonSerializer to serialize your objects

You're not just limited to reading and writing lines of text to your files. You can use **JSON serialization** to let your programs ***copy entire objects*** to strings (which you can write to files!) and read them back in…all in just a few lines of code! Let's take a look at how this works. Start by **creating a new console app**.

Do this!

1 **Design some classes for your object graph.**

Add this HairColor enum and these Guy, Outfit, and HairStyle classes to your new console app:

```
class Guy(string? name, Outfit? clothes, HairStyle? hair) {
    public string? Name => name;
    public HairStyle? Hair => hair;
    public Outfit? Clothes => clothes;
    public override string ToString() => $"{Name} with {Hair} wearing {Clothes}";
}

class Outfit(string? top, string? bottom) {
    public string? Top => top;
    public string? Bottom => bottom;
    public override string ToString() => $"{Top} and {Bottom}";
}

enum HairColor {
    Auburn, Black, Blonde, Blue, Brown, Gray, Platinum, Purple, Red, White
}

class HairStyle(HairColor color, float length) {
    public HairColor Color => color;
    public float Length => length;
    public override string ToString() => $"{Length:0.0} inch {Color} hair";
}
```

You can combine primary constructors with expression-bodied members to make very readable and compact classes.

2 **Create a graph of objects to serialize.**

Now create a small graph of objects to serialize: a new List<Guy> pointing to a couple of Guy objects. Add these top-level statements—they use a collection initializer and object initializers to build the object graph:

```
List<Guy> guys = [
    new Guy(
        "Bob",
        new Outfit("t-shirt", "jeans"),
        new HairStyle(HairColor.Red, 3.5f)
    ),
   new Guy(
       "Joe",
       new Outfit("polo", "slacks"),
       new HairStyle(HairColor.Gray, 2.7f)
   ),
];
```

3 Use JsonSerializer to serialize the objects to a string.

First add a `using` directive to the top of your code file:

```
using System.Text.Json;
```

Visual Studio will add the using statement automatically if you add JsonSerializer to the code.

Now you can **serialize the entire graph** with a single line of code:

```
var jsonString = JsonSerializer.Serialize(guys);
Console.WriteLine(jsonString);
```

Run your app and look closely at what it prints to the console:

```
[{"Name":"Bob","Hair":{"Color":8,"Length":3.5},"Clothes":{"Top":"t-shirt","
Bottom":"jeans"}},{"Name":"Joe","Hair":{"Color":5,"Length":2.7},"Clothes":{-
"Top":"polo","Bottom":"slacks"}}]
```

That's your object graph **serialized to JSON** (which some people pronounce "Jason" and others pronounce "JAY-sahn"). It's a ***human-readable data interchange format***, which means that it's a way to store complex objects using strings that a person can make sense of. Because it's human-readable, you can see that it has all of the parts of the graph: names and clothes are encoded as strings ("Bob", "t-shirt") and enums are encoded as their integer values.

4 Use JsonSerializer to deserialize the JSON to a new object graph.

Now that we have a string that contains the object graph serialized to JSON, we can **deserialize** it. That just means using it to create new objects. JsonSerializer lets us do that in one line of code too. Add this to the top-level statements:

```
var copyOfGuys = JsonSerializer.Deserialize<List<Guy>>(jsonString);
if (copyOfGuys != null)
    foreach (var guy in copyOfGuys)
        Console.WriteLine("I deserialized this guy: {0}", guy);
```

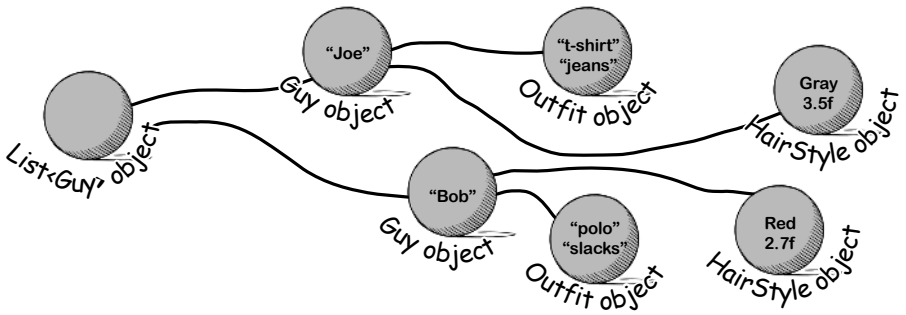
Run your app again. It deserializes the guys from the JSON string and writes them to the console:

```
I deserialized this guy: Bob with 3.5 inch Red hair wearing t-shirt and jeans
I deserialized this guy: Joe with 2.7 inch Gray hair wearing polo and slacks
```

5 Try to spot the object graph in the JSON output.

Here's a picture of the object graph that you created. Look closely at the JSON—can you figure out how it saved the state of all of the objects in the graph?

In this object graph, the List<Guy> is the *root*, or the starting point, which means it's the starting point for the graph and all of the other objects are connected to it.



JSON Up Close

When you use JsonSerializer to serialize an object graph to JSON, it generates a (somewhat) readable text representation of the data in each object. Let's take a closer look at how JSON works.

Go back to your app with the Guy object graph and **replace the line that calls JsonSerializer.Serializer** with this:

```
var options = new JsonSerializerOptions() { WriteIndented = true };
var jsonString = JsonSerializer.Serialize(guys, options);
```

That code calls an overloaded JsonSerializer.Serialize method that takes a JsonSerializerOptions object that lets you set options for the serializer. In this case, you're telling it to write the JSON as indented text—in other words, it adds line breaks and spaces to make the JSON easier for people to read.

Now run the program again. The output should look like this:

```
[
  {
    "Name": "Bob",
    "Hair": {
      "Color": 8,
      "Length": 3.5
    },
    "Clothes": {
      "Top": "t-shirt",
      "Bottom": "jeans"
    }
  },
  {
    "Name": "Joe",
    "Hair": {
      "Color": 5,
      "Length": 2.7
    },
    "Clothes": {
      "Top": "polo",
      "Bottom": "slacks"
    }
  }
]
```

Let's break down exactly what we're seeing:

- The JSON starts and ends with square brackets [ ]. This is how a list is serialized in JSON. A list of numbers would look like this: [1, 2, 3, 4].
- This particular JSON represents a list with two objects. Each object starts and ends with curly braces { }—and if you look at the JSON, you can see that the second line is an opening curly brace `{`, the second-to-last line is a closing curly brace `}`, and in the middle there's a line with `}`, followed by a line with `{`. That's how JSON represents two objects—in this case, the two Guy objects.
- Each object contains a set of keys and values that correspond to the serialized object's properties, separated by commas. For example, `"Name": "Joe",` represents the first Guy object's Name property.
- The Guy.Clothes property is an object reference that points to an Outfit object. It's represented by a nested object with values for Top and Bottom.

IDE Tip: Paste JSON as Classes

The IDE can help you experiment with JSON. When you run your console app, select the output and copy it. Then go back to Visual Studio, add a new class file, and choose **Paste Special >> Paste JSON as Classes** from the menu.

If you have JSON in your copy buffer, Visual Studio will paste a set of class declarations that match the JSON. Try this out, and look closely at what it generates. What classes have names? How does it represent the root of the graph?

(At the time we're writing this, Visual Studio Code does not include the "Paste JSON as Classes" feature.)

There might be a VSCode extension that lets you do this! Go to the extensions window and search for "paste json" and see if there's one available.

JSON only includes data, not specific C# types

When you were looking through the JSON data, you saw human-readable versions of the data in your objects: strings like "Bob" and "slacks", numbers like 8 and 3.5, and even lists and nested objects. What *didn't* you see when you looked at the JSON data? JSON **does not include the names of types**. Look inside a JSON file and you won't see class names like Guy, Outfit, HairColor, or HairStyle, or even the names of basic types like int, string, or double. That's because JSON just contains the data, and JsonSerializer will do its best to deserialize the data into whatever properties it finds.

Let's put this to the test. Add a new class to your project:

```
class Dude(string name, HairStyle hair)
{
    public string Name => name;
    public HairStyle Hair => hair;
}
```

We're deserializing the data from a List of Guy objects into a Stack of Dude objects.

Now add this code to the end of your top-level statements:

```
    var dudes = JsonSerializer.Deserialize<Stack<Dude>>(jsonString);
    while (dudes.Count > 0)
    {
        var dude = dudes.Pop();
        Console.WriteLine($"Next dude: {dude.Name} with {dude.Hair} hair");
    }
```

And run your code again. Since the JSON just has a list of objects, JsonSerializer.Deserialize will happily stick them into a Stack (or a Queue, or an array, or another collection type). Since Dude has public Name and Hair properties that match the data, it will fill in any data that it can. Here's what it prints to the output:

```
Next dude: Joe with 2.7 inch Gray hair hair
Next dude: Bob with 3.5 inch Red hair hair
```

Sharpen your pencil

Let's use JsonSerializer to explore how strings are translated into JSON. Add the following code to a console app, then write down what each line of code writes to the console. The last line serializes the elephant animal emoji.

You used the emoji panel in Chapter 1 to enter emoji.

Console.WriteLine(JsonSerializer.Serialize(3));.....

Console.WriteLine(JsonSerializer.Serialize((long)-3)); .....

Console.WriteLine(JsonSerializer.Serialize((byte)0)); .....

Console.WriteLine(JsonSerializer.Serialize(float.MaxValue)); .....

Console.WriteLine(JsonSerializer.Serialize(float.MinValue)); .....

Console.WriteLine(JsonSerializer.Serialize(true)); .....

Console.WriteLine(JsonSerializer.Serialize("Elephant")); .....

Console.WriteLine(JsonSerializer.Serialize("Elephant".ToCharArray())); .....

Console.WriteLine(JsonSerializer.Serialize("🐘")); .....

One more thing! We showed you basic serialization with JsonSerializer. There are just a couple more things you need to know about it.

Watch it!

JsonSerializer only serializes public properties (not fields), and requires a parameterless constructor.

Remember the SwordDamage class from Chapter 5? Its Damage property has a private set accessor:

```
public int Damage { get; private set; }
```

It also has a constructor that takes an int parameter:

```
public SwordDamage(int startingRoll)
```

You'll be able to use JsonSerializer to serialize a SwordDamage object without any problems. If you try to deserialize one, JsonSerializer will throw an exception—at least, it will if you use the code we've shown you. If you want to serialize objects that save their state in fields, or private properties, or use constructors with parameters, you'll need to create a converter.

You can read more about how that works in the .NET serialization documentation: https://docs.microsoft.com/dotnet/standard/serialization

Bullet Points

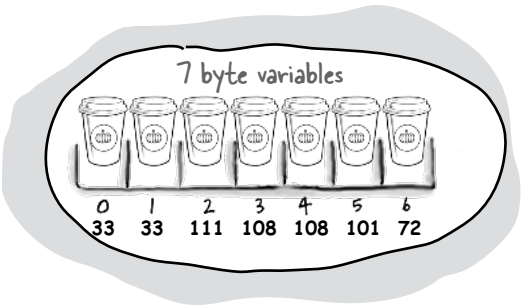
- **Serialization** means writing an object's entire state to a file or string. **Deserialization** means reading the object's state back from that file or string.
- A group of objects connected to each other by references is sometimes referred to as a **graph**.
- When an object is serialized, the **entire graph** of objects it refers to is serialized along with it so they can all be deserialized together.
- The **JsonSerializer class** has a static Serialize method that serializes an object graph to JSON, and a static Deserialize method that instantiates an object graph using serialized JSON data.
- JSON data is **human-readable** (for the most part). Values are serialized as plain text: strings are written in "quotes" and other literals (like numbers and Boolean values) are encoded without quotes.
- JSON represents **arrays** of values using square brackets [ ].
- JSON represents **objects** inside curly brackets { }, with members and their values represented as key/value pairs separated by a colon.
- JSON **does not store specific types** like string or int, or specific class names. Instead, it relies on "smart" classes like JsonSerializer to do their best to match the data to the type that the data is being deserialized into.

Next up: we'll take a deep dive into our data

You've been writing lots of code using value types like int, bool, and double, and creating objects that store data in fields. Now it's time to take a lower-level view of things. The rest of this chapter is all about getting to know your data better by understanding the actual bytes that C# and .NET use to represent it.

Here's what we'll do.

.NET uses Unicode to store characters and text. We'll explore how C# strings are encoded with Unicode.



We'll write values as binary data, then we'll read them back in and see what bytes were written.

```
0000: 45 6c 65 6d 65 6e 74 61 Elementa
0005: 72 79 2c 20 6d 79 20 64 ry, my d
0010: 65 61 72 20 57 61 74 73 ear Wats
0015: 6f 6e 21                on!
```

We'll build a hex dumper that lets us take a closer look at the bits and bytes in files.

Game Design...and Beyond

Accessibility

You've been learning about accessibility throughout this book. Did you know it's also an important part of game design? Think about it: how do you play your favorite game if you're visually impaired? What if you have a hearing impairment, or a motor or dexterity impairment that makes it difficult for you to hold a controller? Game accessibility is all about making our games accessible to everyone, regardless of any disabilities or impairments.

Wait, what? There are blind gamers? Yes! Did you think that the "video" part of video games means they're inaccessible to people with visual impairments? Take a few minutes to search YouTube for "blind gamer" and watch videos of people with visual impairments—including folks who have partial or complete blindness!—showcase serious gaming skills.

One important thing you can do as a developer to make your game accessible is to take time to ***understand players with impairments***. What did you learn from watching those videos?

Something we learned from watching blind gamers is that they use ***sounds from the games*** to understand what's going on. In a fighting game, different moves may make different sounds. In a platformer, enemies coming toward the player might make telltale chirps or clicks. We also learned that while some games provide adequate ***audio cues***, others don't—and few (if any) games are designed to give blind players adequate audio cues to play the game.

Many gamers have ***hearing impairments***, so games that have audio cues should also have visual ones. Try playing your game with the sound on mute. Does that keep important information from being conveyed to the player? Are there visual cues that go along with the audio? Are there subtitles for all of the dialog? Make sure your game is playable without sound.

One in 12 men and 1 in 200 women have some form of ***color blindness*** (including one of the authors of this book!). Many high-budget games include a color-blind mode that does sophisticated color adjustments. You can make your game more accessible to color-blind people by using colors that have a ***high contrast*** with each other.

Many gamers have a wide range of ***motor impairments***, from repetitive strain injuries to paralysis. Players who are unable to use conventional input devices like a mouse and keyboard or a controller may use ***assistive hardware*** like an eye tracker or modified controller. One way you can accommodate these players is to make it easy for your games to accept keyboard mappings by letting players set up a profile where different keys map to the game controls.

You may have noticed that many games start with a warning screen about seizures. That's because many gamers with ***epilepsy*** are photosensitive, which means that certain flashing or flickering patterns can cause them to have seizures. While seizure warnings are important, we can do better. As developers, it's important for us to put effort into understanding and avoiding the kinds of strobe lights, flashing, and other visual patterns most likely to trigger seizures. Take the time to read this editorial by video game reviewer Cathy Vice about her ***experience as a gamer with epilepsy***: *https://indiegamerchick.com/2013/08/06/the-epilepsy-thing*

Accessibility is abbreviated as #a11y—that's a "numeronym" that means "A" followed by 11 letters ("ccessibilit") followed by "Y." It's a great way to remind us that we can all be responsible allies.

C# strings are encoded with Unicode

You've been using strings since you typed `"Hello, world!"` into the IDE at the start of Chapter 1. Because strings are so intuitive, we haven't really needed to dissect them and figure out what makes them tick. But ask yourself…***what exactly is a string?***

A C# string is a **read-only collection of chars**. So if you look at how a string is stored in memory, you'll see the string "Elephant" stored as the chars 'E', 'l', 'e', 'p', 'h', 'a', 'n', and 't'. Now ask yourself…***what exactly is a char?***

A char is a character represented with **Unicode**. Unicode is an industry standard for **encoding** characters, or converting them into bytes so they can be stored in memory, transmitted across networks, included in documents, or pretty much anything else you want to do with them—and you're guaranteed that you'll always get the correct characters.

This is especially important when you consider just how many characters there are. The Unicode standard supports over 150 **scripts** (sets of characters for specific languages), including not just Latin (which has the 26 English letters and variants like é and ç) but scripts for many languages used around the world. The list of supported scripts is constantly growing, as the Unicode Consortium adds new ones every year (here's the current list: *http://www.unicode.org/standard/supported.html*).

Unicode supports another really important set of characters: **emoji**. All of the emoji, from the winking smiley face (😜) to the ever-popular pile of poo (💩), are Unicode characters.

Your animal matching game in Chapter 1 treated emoji characters like any other C# chars.

Watch it!

Assistive technology can be thrown off by Unicode characters.

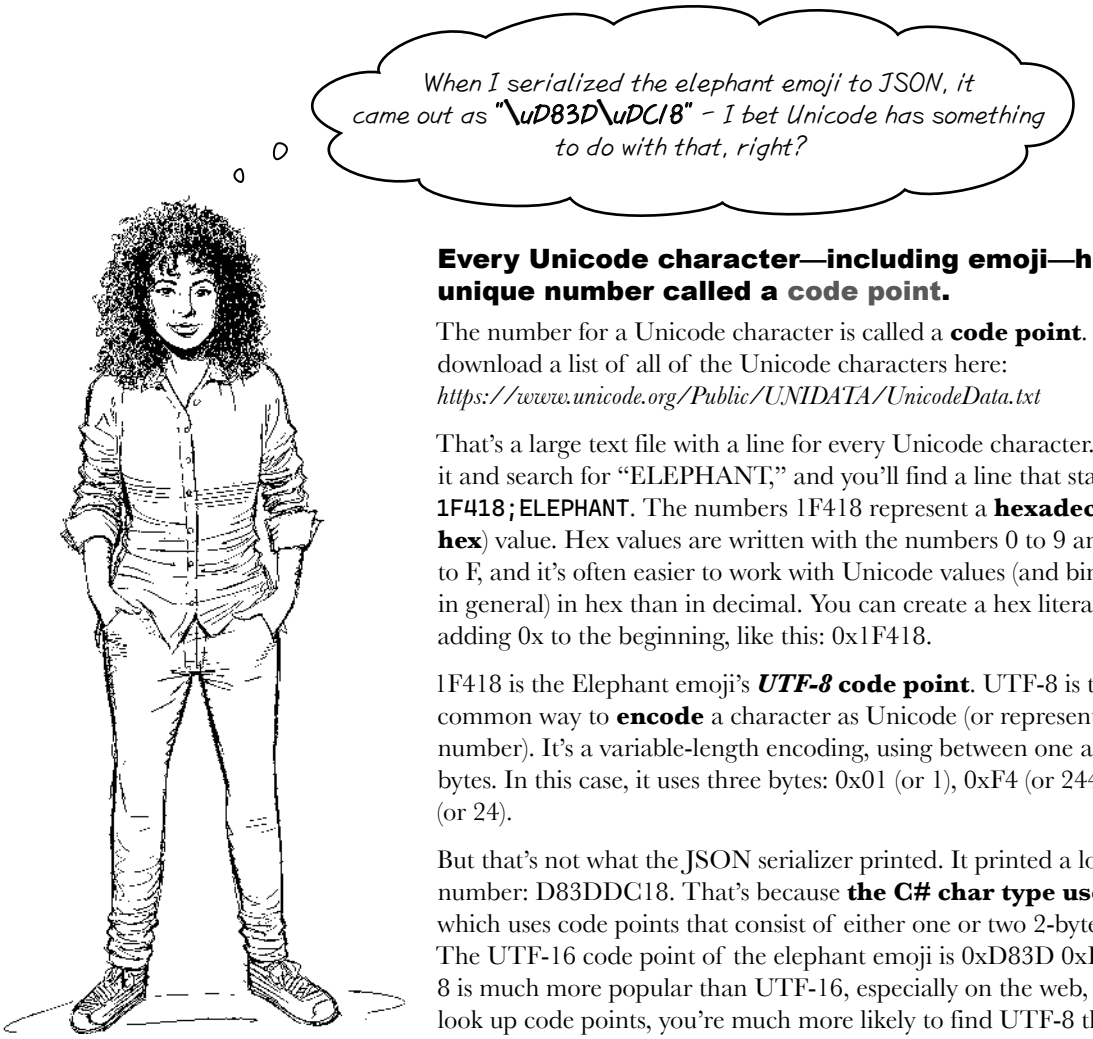
Accessibility is extremely important. We felt that it's especially valuable to bring up accessibility here because—like many of our "Game design...and beyond" topics—we can use it to teach a lesson that applies to all software development. You've probably seen posts on social media that use emoji or other "funny-looking" characters, and many that use boldfaced, cursive, or upside-down characters. On some platforms, these are done with Unicode characters, and when done like that they can be very problematic for assistive technology.

Some folks make social media posts like this: I'm💣using👏hand👏claps👏to👏emphasize👏points

On-screen, those hand-clap characters look fine. But a screen reader like Windows Narrator or macOS VoiceOver might read that message aloud like this: "I'm clapping hands using clapping hands claps clapping hands to clapping hands emphasize clapping hands points."

You might see "fonts" that look like this: 𝔱𝔥𝔦𝔰 𝔦𝔰 𝒶 Ⓜ𝒆Ⓢ5ɑGE 𝒊𝒏 𝒶 really ɹǝlɐʍ FONT

A screen reader will read 𝔪 *as "mathematical bold Fraktur small m" and other characters as "script letter n" or "mathematical double stroke small a," or just leave them out entirely. The experience could be just as bad for someone using a braille reader. Are those assistive technologies somehow broken? Not at all—they're doing their job. Those are the real names for those Unicode characters, and the assistive technologies are accurately describing the text. As you go through the next part of this chapter, keep these examples in mind—they'll help you better understand how Unicode works.*



Every Unicode character—including emoji—has a unique number called a code point.

The number for a Unicode character is called a **code point**. You can download a list of all of the Unicode characters here: *https://www.unicode.org/Public/UNIDATA/UnicodeData.txt*

That's a large text file with a line for every Unicode character. Download it and search for "ELEPHANT," and you'll find a line that starts like this: `1F418;ELEPHANT`. The numbers 1F418 represent a **hexadecimal** (or **hex**) value. Hex values are written with the numbers 0 to 9 and letters A to F, and it's often easier to work with Unicode values (and binary values in general) in hex than in decimal. You can create a hex literal in C# by adding 0x to the beginning, like this: 0x1F418.

1F418 is the Elephant emoji's ***UTF-8* code point**. UTF-8 is the most common way to **encode** a character as Unicode (or represent it as a number). It's a variable-length encoding, using between one and four bytes. In this case, it uses three bytes: 0x01 (or 1), 0xF4 (or 244), and 0x18 (or 24).

But that's not what the JSON serializer printed. It printed a longer hex number: D83DDC18. That's because **the C# char type uses *UTF-16***, which uses code points that consist of either one or two 2-byte numbers. The UTF-16 code point of the elephant emoji is 0xD83D 0xDC18. UTF-8 is much more popular than UTF-16, especially on the web, so when you look up code points, you're much more likely to find UTF-8 than UTF-16.

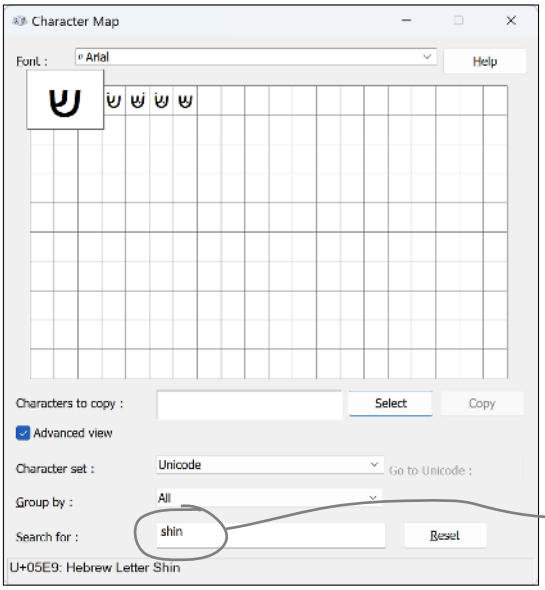
UTF-8 is a variable-length encoding used by most web pages and many systems. It can store characters using one, two, three, or more bytes. UTF-16 is a fixed-length encoding that always uses either one or two 2-byte numbers. .NET stores char values in memory as UTF-16 values.

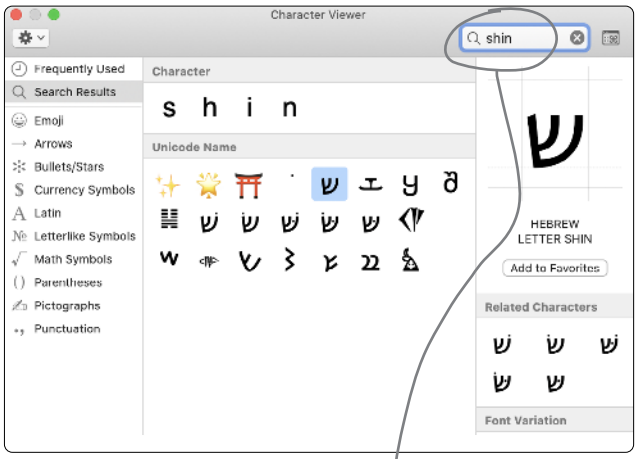
Visual Studio works really well with Unicode

Let's use Visual Studio to see how the IDE works with Unicode characters. You saw back in Chapter 1 that you can use emoji in code. Let's see what else the IDE can handle. Go to the code editor and enter this code:

```
Console.WriteLine("Hello ");
```

If you're using Windows, open up the Character Map app. If you're using a Mac, press the fn key (or Ctrl+⌘+Space on an older Mac) to pop up the Character Viewer. Then search for the Hebrew letter shin (ש) and copy it to the clipboard.





Both the Windows Character Map app and the macOS Character Viewer let you search for Unicode characters and copy them to the clipboard.

Place your cursor at the end of the string between the space and the quotation mark, and paste in the shin character that you copied to the clipboard. Hmm, something looks weird:

```
interface System.IDisposable
```

Did you notice that the cursor is positioned to the *left* of the pasted letter? Well, let's continue. Don't click anywhere in the IDE—keep the cursor where it is, then switch over to Character Map or Character Viewer to search for the Hebrew letter lamed (ל). Switch back to the IDE—make sure the cursor is still positioned just left of the shin—and paste in the lamed:

```
Console.WriteLine("Hello של");
```

When you pasted the lamed, the IDE added it to the left of the shin. Now search for the Hebrew letters vav (ו) and final mem (ם). Paste each of them into the IDE—it will insert them to the left of the cursor:

```
Console.WriteLine("Hello שלום");
```

The IDE knows that ***Hebrew is read right to left***, so it's behaving accordingly. Click to select the text near the beginning of the statement, and slowly drag your cursor right to select `Hello` and then שלום. Watch carefully what happens when the selection reaches the Hebrew letters. It skips to the shin (ש) and then selects from right to left—and that's exactly what a Hebrew reader would expect it to do.

.NET uses Unicode to store characters and text

The two C# types for storing text and characters—string and char—keep their data in memory as Unicode. When that data is written out as bytes to a file, each of those Unicode numbers is written out to the file. Let's get a sense of exactly how Unicode data is written out to a file. **Create a new console app.** We'll use the File.WriteAllBytes and File.ReadAllBytes methods to start exploring Unicode.

Do this!

1 **Write a normal string out to a file and read it back.**

Add the following code to the top-level statements—it uses File.WriteAllText to write the string "Eureka!" out to a file called *eureka.txt*. Then it creates a new byte array called `eurekaBytes`, reads the file into it, and prints out all of the bytes it read:

```
File.WriteAllText("eureka.txt", "Eureka!");
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");
foreach (byte b in eurekaBytes)
    Console.Write("{0} ", b);
Console.WriteLine(Encoding.UTF8.GetString(eurekaBytes));
```

The ReadAllBytes method returns a reference to a new array of bytes that contains all of the bytes that were read in from the file.

You'll see these bytes written to the output: `69 117 114 101 107 97 33`. The last line calls the method Encoding.UTF8.GetString, which converts a byte array with UTF-8-encoded characters to a string. Now **open the file in Notepad** (Windows) **or TextEdit** (Mac). It says "Eureka!"

2 **Then add code to write the bytes as hex numbers.**

When you're encoding data you'll often use hex, so let's do that now. Add this code to the end of the top-level statements to write the same bytes out, using `{0:x2}` to **format each byte as a hex number**:

```
foreach (byte b in eurekaBytes)
    Console.Write("{0:x2} ", b);
Console.WriteLine();
```

Hex uses the numbers 0 through 9 and letters A through F to represent numbers in base 16, so 6B is equal to 107.

That tells Write to print parameter `0` (the first one after the string to print) as a two-character hex code. So it writes the same seven bytes in hex instead of decimal: `45 75 72 65 6b 61 21`.

3 **Modify the first line to write the Hebrew letters "שלום" instead of "Eureka!"**

You just added the Hebrew text שלום to another program using either Character Map (Windows) or Character Viewer (Mac). **Comment out the first line of the top-level statements and replace it with the following code** that writes **"שלום"** to the file instead of "Eureka!" We've added an extra Encoding.Unicode parameter so it writes UTF-16 (remember, the Encoding class is in the System.Text namespace, so make sure you *also add* `using System.Text;` to the top):

```
File.WriteAllText("eureka.txt", "שלום", Encoding.Unicode);
```

Now run the code again, and look closely at the output: `ff fe e9 05 dc 05 d5 05 dd 05`. The first two characters are "FF FE", which is the Unicode way of saying that we're going to have a string of 2-byte characters. The rest of the bytes are the Hebrew letters—but they're reversed, so U+05E9 appears as `e9 05`. Now open the file up in Notepad or TextEdit to make sure it looks right.

4 **Use JsonSerializer to explore UTF-8 and UTF-16 code points.**

When you serialized the elephant emoji, JsonSerializer generated `\uD83D\uDC18`—which we now know is the 4-byte UTF-16 code point in hex. Now let's try that with the Hebrew letter shin. Add `using System.Text.Json;` to the top of your app and then add this line:

```
Console.WriteLine(JsonSerializer.Serialize("ש"));
```

Run your app again. This time it printed a value with two hex bytes, "\u05E9"—that's the UTF-16 code point for the Hebrew letter shin. It's also the UTF-8 code point for the same letter.

But wait a minute—we learned that the UTF-8 code point for the elephant emoji is `0x1F418`, which is ***different*** than the UTF-16 code point (0xD83D 0xDC18). What's going on?

It turns out that most of the characters with 2-byte UTF-8 code points have the same code points in UTF-16. Once you reach the UTF-8 values that require three or more bytes—which includes the familiar emoji that we've used in this book—they differ. So while the Hebrew letter shin is 0x05E9 in both UTF-8 and UTF-16, the elephant emoji is 0x1F418 in UTF-8 and 0xD8ED 0xDC18 in UTF-16.

Use \u escape sequences to include Unicode in strings

When you serialized the elephant emoji, JsonSerializer generated \uD83D\uDC18—the 4-byte UTF-16 code point for the emoji in hex. That's because both JSON and C# strings use **UTF-16 escape sequences**—and it turns out JSON uses the same escape sequences.

Characters with 2-byte code points like ש are represented with a \u followed by the hex code point (\u05E9); characters with 4-byte code points like 🐘 are represented with \u and the highest two bytes, followed by \u and the lowest two bytes (\uD83D\uDC18).

C# also has another Unicode escape sequence: \U (with an UPPERCASE U) followed by eight hex bytes lets you embed a **UTF-32 code point**, which is always four bytes long. That's yet another Unicode encoding, and it's really useful because you can convert UTF-8 to UTF-32 by just padding the hex number with zeros—so the UTF-32 code point for ש is \U00000 5E9, and for 🐘 it's \U0001F418.

5 **Use Unicode escape sequences to encode 🐘.**

Add these lines to your app to write the elephant emoji to two files using both the UTF-16 and UTF-32 escape sequences:

```
File.WriteAllText("elephant1.txt", "\uD83D\uDC18");
File.WriteAllText("elephant2.txt", "\U0001F418");
```

Run your app again, then open both of those files in Notepad or TextEdit. You should see the correct character written to the file.

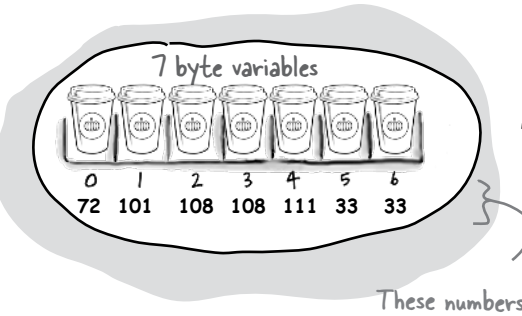
You used UTF-16 and UTF-32 escape sequences to create your emoji, but the WriteAllText method *writes a UTF-8 file*. The Encoding.UTF8.GetString method you used in step 1 converts a byte array with UTF-8-encoded data back to a string.

C# can use byte arrays to move data around

Since all your data ends up encoded as **bytes**, it makes sense to think of a file as one **big byte array**…and you already know how to read and write byte arrays.

Here's the code to create a byte array, open an input stream, and read the text "Hello!!" into bytes 0 through 6 of the array.

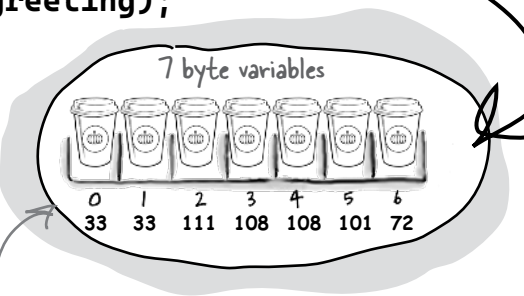
```
byte[] greeting;
greeting = File.
ReadAllBytes(filename);
```



These numbers are the Unicode values for the characters in "Hello!!"

This is a static method in the Array class that reverses the order of the bytes. We're just using it to show that the changes you make to the byte array get written out to the file exactly.

```
Array.Reverse(greeting);
File.WriteAllBytes(filename,
greeting);
```



Now the bytes are in reverse order.

When the program writes the byte array out to a file, the text is in reverse order too.

Reversing the bytes in "Hello!!" only works because each of those characters is one byte long. Can you figure out why this won't work for שלום?

Use a BinaryWriter to write binary data

StreamWriter also encodes your data. It just specializes in text and text encoding—it defaults to UTF-8.

You ***could*** encode all of your strings, chars, ints, and floats into byte arrays before writing them out to files, but that would get pretty tedious. That's why .NET gives you a very useful class called **BinaryWriter** that **automatically encodes your data** and writes it to a file. All you need to do is create a FileStream and pass it into the BinaryWriter's constructor (they're in the System.IO namespace, so you'll need `using System.IO;`). Then you can call its methods to write out your data. Let's practice using BinaryWriter `to write binary data to a file`.

Do this!

1. Start by creating a console app and setting up some data to write to a file:

```
int intValue = 48769414;
string stringValue = "Hello!";
byte[] byteArray = { 47, 129, 0, 116 };
float floatValue = 491.695F;
char charValue = 'E';
```

If you use File.Create, it'll start a new file—if there's one there already, it'll blow it away and start a brand-new one. The File.OpenWrite method opens the existing one and starts overwriting it from the beginning instead.

2. To use a BinaryWriter, first you need to open a new stream with File.Create:

```
using (var output = File.Create("binarydata.dat"))
using (var writer = new BinaryWriter(output))
{
```

3. Now just call its Write method. Each time you do this, it adds new bytes onto the end of the file that contain an encoded version of whatever data you passed it as a parameter:

```
    writer.Write(intValue);
    writer.Write(stringValue);
    writer.Write(byteArray);
    writer.Write(floatValue);
    writer.Write(charValue);
}
```

Each Write statement encodes one value into bytes, and then sends those bytes to the FileStream object. You can pass it any value type, and it'll encode it automatically.

The FileStream writes the bytes to the end of the file.

Sharpen your pencil

4. Now use the same code you used before to read in the file you just wrote:

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.Write("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);
```

Write down the output in the blanks below. Can you **figure out what bytes correspond** to each of the five `writer.Write(...)` statements? We put a bracket under the groups of bytes that correspond with each statement to help you figure out which bytes in the file correspond with data written by the app.

Here's a hint: strings can be different lengths, so the string has to start with a number to tell .NET how long it is. BinaryWriter uses UTF-8 to encode strings, and in UTF-8 all of the characters in "Hello!" have UTF code points that consist of a single byte. Download UnicodeData.txt from unicode.org (we gave you the URL earlier) and use it to look up the code points for each character.

__ - ___ bytes

intValue stringValue byteArray floatValue charValue

Use BinaryReader to read the data back in

The BinaryReader class works just like BinaryWriter. You create a stream, attach the BinaryReader object to it, and then call its methods…but the reader ***doesn't know what data's in the file!*** It has no way of knowing. Your float value of 491.695F was encoded as d8 f5 43 45. Those same bytes are a perfectly valid int—1,140,185,334—so you'll need to tell the BinaryReader exactly what types to read from the file. Add the following code to your program, and have it read the data you just wrote.

Don't take our word for it. Replace the line that reads the float with a call to ReadInt32. (You'll need to change the type of floatRead to int.) Then you can see for yourself what it reads from the file.

1. Start out by setting up the FileStream and BinaryReader objects:

```
using (var input = File.OpenRead("binarydata.dat"))
using (var reader = new BinaryReader(input))
{
```

2. You tell BinaryReader what type of data to read by calling its different methods:

```
    int intRead = reader.ReadInt32();
    string stringRead = reader.ReadString();
    byte[] byteArrayRead = reader.ReadBytes(4);
    float floatRead = reader.ReadSingle();
    char charRead = reader.ReadChar();
```

Each value type has its own method in BinaryReader that returns the data in the correct type. Most don't need any parameters, but ReadBytes takes one parameter that tells BinaryReader how many bytes to read.

3. Now write the data that you read from the file to the console:

```
    Console.Write("int: {0}  string: {1}  bytes: ", intRead, stringRead);
    foreach (byte b in byteArrayRead)
        Console.Write("{0} ", b);
    Console.Write(" float: {0}  char: {1} ", floatRead, charRead);
}
```

Here's the output that gets printed to the console:

```
int: 48769414  string: Hello!  bytes: 47 129 0 116  float: 491.695  char: E
```

Sharpen your pencil Solution

float and int values take up four bytes when you write them to a file. If you'd used long or double, then they'd take up eight bytes each.

86 29 e8 02 06 48 65 6c 6c 6f 21 2f 81 00 74 f6 d8 f5 43 45 - 20 bytes

intValue | stringValue | byteArray | floatValue | charValue

The first byte in the string is 6—that's the length of the string. You can use Character Map to look up each of the characters in "Hello!"—it starts with U+0048 and ends with U+0021.

Both the Windows and Mac calculator apps have programmer modes that can convert these bytes from hex to decimal, which will let you convert them back to the values in the array.

char holds a Unicode character, and "E" only takes one byte—it's encoded as U+0045.

there are no Dumb Questions

Q: Earlier in the chapter when I wrote "Eureka!" to a file and then read the bytes back, it took one byte per character. So why did each of the Hebrew letters in שלום take two bytes? And why did it write the bytes "FF FE" at the beginning of the file?

A: What you're seeing is the difference between two closely related Unicode encodings. Latin characters (including plain English letters), numbers, normal punctuation marks, and some standard characters (like curly brackets, ampersands, and other things you see on your keyboard) all have very low Unicode numbers—between 0 and 127. They correspond to a very old encoding called ASCII that dates back to the 1960s, and UTF-8 was designed to be backward compatible with ASCII. A file with only those Unicode characters contains just their bytes and nothing else.

Things get a little more complicated when you add Unicode characters with higher-numbered code points into the mix. One byte can only hold a number between 0 and 255. Two bytes in a row can store numbers between 0 and 65,536—which, in hex, is FFFF. The file needs to be able to tell whatever program opens it up that it's going to contain these higher-numbered characters, so it puts a special reserved byte sequence at the beginning of the file: FF FE. That's called the ***byte order mark***. As soon as a program sees that, it knows that all of the characters are encoded with two bytes each (so an E is encoded as 00 45, with a leading zero).

Q: Why is it called a byte order mark?

A: Go back to the code that wrote שלום to a file, then printed the bytes it wrote. You'll see that the bytes in the file were reversed. For example, the ש code point U+05E9 was written to the file as E9 05. That's called *little-endian*—it means the least significant byte is written first. Go back to the code that calls WriteAllText and modify it to **change the third argument from Encoding.Unicode to Encoding.BigEndianUnicode**. That tells it to write the data out in *big-endian*, which doesn't flip the bytes around—when you run it again, you'll see the bytes come out as 05 E9 instead. You'll also see a different byte order mark: FE FF. This tells Notepad or TextEdit how to interpret the bytes in the file.

Q: Why didn't I use a `using` block or call Close after I used File.ReadAllText and File.WriteAllText?

A: The File class has several very useful static methods that automatically open up a file, read or write data, and then **close it automatically**. In addition to the ReadAllText and WriteAllText methods, there are ReadAllBytes and WriteAllBytes, which work with byte arrays, and ReadAllLines and WriteAllLines, which read and write string arrays, where each string in the array is a separate line in the file. All of these methods automatically open and close the streams, so you can do your whole file operation in a single statement.

Q: If FileStream has methods for reading and writing, why do I ever need to use StreamReader and StreamWriter?

A: The FileStream class is really useful for reading and writing bytes in binary files. Its methods for reading and writing operate with bytes and byte arrays. A lot of programs work exclusively with text files, and that's where StreamReader and StreamWriter come in really handy. They have methods that are built specifically for reading and writing lines of text. Without them, if you wanted to read a line of text from a file, you'd have to first read a byte array and then write a loop to search through that array for a linebreak—so it's not hard to see how they make your life easier.

If you're writing a string that only has Unicode characters with low numbers (such as Latin letters), it writes one byte per character. If it's got high-numbered characters (like emoji characters), they'll be written using two or more bytes each.

A hex dump lets you see the bytes in your files

A **hex dump** is a *hexadecimal* view of the contents of a file, and it's a really common way for programmers to take a deep look at a file's internal structure.

It turns out that hex is a convenient way to display bytes in a file. A byte takes two characters to display in hex: bytes range from 0 to 255, or 00 to ff in hex. That lets you see a lot of data in a really small space, and in a format that makes it easier to spot patterns. It's useful to display binary data in rows that are 8, 16, or 32 bytes long because most binary data tends to break down in chunks of 4, 8, 16, or 32…like all the types in C#. (For example, an int takes up 4 bytes.) A hex dump lets you see exactly what those values are made of.

How to make a hex dump of some plain text

Start with some familiar text using Latin characters:

```
When you have eliminated the impossible, whatever remains, however
improbable, must be the truth. - Sherlock Holmes
```

First, break up the text into 16-character segments, starting with the first 16: When you have el

Next, convert each character in the text to its UTF-8 code point. Since the Latin characters all have ***1-byte*** UTF-8 code points, each will be represented by a two-digit hex number from 00 to 7F. Here's what each line of our dump will look like:

This is the segment's offset (or position in the file) written as a hex number.

These are the first 8 bytes of the 16-byte segment.

This divider makes the line more readable.

These are the last 8 bytes of the 16-byte segment.

These are the text characters that were dumped.

```
0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c    When you have el
```

Repeat until you've dumped every 16-character segment in the file:

```
0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c    When you have el
0010: 69 6d 69 6e 61 74 65 64 -- 20 74 68 65 20 69 6d 70    iminated the imp
0020: 6f 73 73 69 62 6c 65 2c -- 20 77 68 61 74 65 76 65    ossible, whateve
0030: 72 20 72 65 6d 61 69 6e -- 73 2c 20 68 6f 77 65 76    r remains, howev
0040: 65 72 20 69 6d 70 72 6f -- 62 61 62 6c 65 2c 20 6d    er improbable, m
0050: 75 73 74 20 62 65 20 74 -- 68 65 20 74 72 75 74 68    ust be the truth
0060: 2e 20 2d 20 53 68 65 72 -- 6c 6f 63 6b 20 48 6f 6c    . - Sherlock Hol
0070: 6d 65 73 0a             --                            mes.
```

And that's our dump. There are many hex dump programs for various operating systems, and each of them has a slightly different output. Each line in our particular hex dump format represents 16 characters in the input that was used to generate it, with the offset at the start of each line and the text for each character at the end. Other hex dump apps might display things differently (for example, rendering escape sequences or showing values in decimal).

A hex dump is a hexadecimal view of data in a file or memory, and can be a really useful tool to help you debug binary data.

Use StreamReader to build a hex dumper

Let's build a hex dump app that reads data from a file with StreamReader and writes its dump to the console. We'll use the **StreamReader.ReadBlock method**, which reads characters into a char[] array. It there aren't enough characters left to fill up the array, it fills the rest of the buffer up with the **`'\0'` null character**, which is a Unicode character (sometimes abbreviated as NUL) that represents the end of a string.

The *ReadBlock method* reads the next characters from its input into a byte array (sometimes referred to as a **buffer**). It *blocks*, which means it keeps executing and doesn't return until it's either read all of the characters you asked for or run out of data to read.

Create a new console app called HexDump. Here's the code for the app—it writes a line of text to a file called *textdata.txt*, then opens the file up, reads its contents 16 characters at a time, and writes a hex dump to the console.

```
File.WriteAllText("textdata.txt", "Elementary, my dear Watson!");

using (var reader = new StreamReader("textdata.txt"))
{
    var position = 0;
    while (!reader.EndOfStream)
    {
        // Read up to the next 16 bytes from the file into a buffer
        var buffer = new char[16];
        var bytesRead = reader.ReadBlock(buffer);

        // Write the position (or offset) in hex, followed by a colon and space
        Console.Write("{0:x4}: ", position);
        position += bytesRead;

        // Write the hex value of each character in the byte array
        for (var i = 0; i < 16; i++)
        {
            if (i < bytesRead)
                Console.Write("{0:x2} ", (byte)buffer[i]);
            else
                Console.Write("   ");
            if (i == 7) Console.Write("-- ");
        }

        // Write the actual characters in the buffer
        var bufferContents = new String(buffer);
        Console.WriteLine("   {0}", bufferContents);
    }
}
```

A StreamReader's EndOfStream property returns false if there are characters still left to read in the file.

The {0:x4} formatter converts a numeric value to a four-digit hex number, so 1984 is converted to "07c0".

You can use the *String constructor* to create a string from a char[] array. It will convert the whole array to a string unless it finds a '\0' null character, in which case it will end the string before that character.

Now run your app. It will print a hex dump to the console:

```
0000: 45 6c 65 6d 65 6e 74 61 -- 72 79 2c 20 6d 79 20 64    Elementary, my d
0010: 65 61 72 20 57 61 74 73 -- 6f 6e 21                   ear Watson!
```

Take the time to really understand how this code works. Pay attention to the `for` loop, which goes through the characters and prints each of them as hex in the output.

Use Stream.Read to read bytes from a stream

The hex dumper works just fine for simple text files—but there's a problem. Let's use the hex dumper to read the binary data that you wrote earlier in the chapter. Copy the code that you wrote earlier to the top of your app, then change the name of the file that it reads to binarydata.dat:

```
using (var output = File.Create("binarydata.dat"))
using (var writer = new BinaryWriter(output))
{
    writer.Write(48769414);
    writer.Write("Hello!");
    writer.Write(new byte[] { 47, 129, 0, 116 });
    writer.Write(491.695F);
    writer.Write('E');
}

using (var reader = new StreamReader("binarydata.dat"))
{
   // the rest of the code for the app stays the same
```

Add this code—it's almost identical to the code you wrote earlier to write binary values to a file called binarydata.dat.

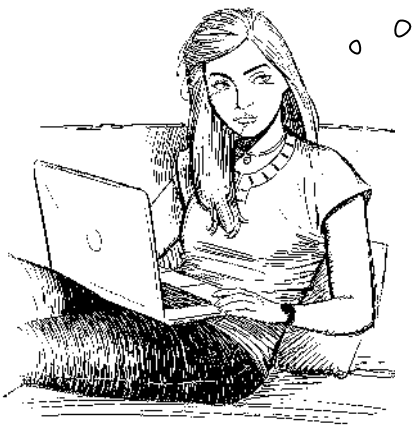
Make your code read the binary data it just wrote.

Now run your app. This time it prints something else—but it's not quite right:

```
0000: fd 29 fd 02 06 48 65 6c -- 6c 6f 21 2f fd 00 74 fd    ?)?Hello!/? t?
0010: fd fd 43 45             --                            ??CE
```

These bytes were 81 and f6 in the "Sharpen your pencil" solution, but StreamReader changed them to fd.

The text characters ("Hello!") seem OK. But compare the output with the "Sharpen your pencil" solution—the bytes aren't quite right. It looks like it replaced some bytes (86, e8, 81, f6, d8, and f5) with a different byte, fd. That's because ***StreamReader is built to read text files***.



Why does it matter whether a file has ***text data or binary data****? It's all just bytes, right?*

Text data and binary data use different encodings.

A text file contains only text characters. StreamReader can read UTF-8, UTF-16, and Unicode characters—but only if the file has a **byte order mark**, or special characters at the start of the file that indicate what encoding it's using. When StreamReader sees a byte order mark, it starts reading the file *immediately* ***after*** *the byte order mark*. Your dumper reads each character and casts it to a byte. If the file is UTF-16, each character (represented by two or more bytes) is turned into one byte in the output, so the output it prints will skip a lot of bytes.

If StreamReader doesn't find a byte order mark, it assumes the file only contains **7-bit data**, or text characters with byte values between 0 and 127 (which is 1111111 in binary). When it encounters a character that it doesn't recognize, it replaces it with the **replacement character**, or Unicode U+FFFD (or just FD in a file with single-byte characters). It's typically represented with a ⍰ symbol. You've probably seen that symbol in a web browser when it encounters a page with data that it can't display.

Modify your hex dumper to read directly from the stream

Your hex dumper uses StreamReader, and that's keeping it from being able to read a lot of characters that appear in files. Your app does use a stream, but only *indirectly* because it's encapsulated inside the StreamReader object. We've seen how StreamReader has problems reading binary data: if it doesn't find a byte order mark indicating that it's reading text from a text file, it only reads 7-bit values. If it does read a text file, it skips over the byte order mark at the beginning and starts reading characters. The purpose of a hex dumper is to show you the actual bytes in a file, so StreamReader is not a good choice to implement it.

Let's do this right—by ***reading the bytes directly from the stream***. Modify the `using` block so it uses **File.OpenRead**, which opens the file and **returns a FileStream**.

The FileStream object implements the **Stream.Read method**, which reads a specified number of bytes from a stream into a byte array buffer. If there are no more bytes to read, it returns 0—in which case we'll end the execution by calling **`return`**.

You'll use the Stream's Length property to keep reading until you've read all of the bytes in the file, and its Read method to read the next 16 bytes into the byte array buffer:

```
using (Stream input = File.OpenRead("binarydata.dat"))
{
    var position = 0;
    var buffer = new byte[16];
    while (true)
    {
        // Read up to the next 16 bytes from the file into a buffer
        var bytesRead = input.Read(buffer, 0, buffer.Length);
        if (bytesRead == 0) return;

        // Write the position (or offset) in hex, followed by a colon and space
        Console.Write("{0:x4}: ", position);
        position += bytesRead;

        // Write the hex value of each character in the byte array
        for (var i = 0; i < 16; i++)
        {
            if (i < bytesRead)
                Console.Write("{0:x2} ", (byte)buffer[i]);
            else
                Console.Write("   ");
            if (i == 7) Console.Write("-- ");
        }

       // Write the actual characters in the buffer
       for (int i = 0; i < bytesRead; i++)
           Console.Write((char)buffer[i]);
       Console.WriteLine();
   }
}
```

This is the same File.OpenRead method you used earlier to get a stream with the contents of a file.

This is where the app calls Stream.Read to read bytes from the file.

This code is exactly the same as in the previous version of the hex dumper that used StreamReader.

Instead of passing the entire buffer to Console.WriteLine, now you're using a for loop to write each individual byte to the console.

Now it prints values that match the "Sharpen your pencil" solution:

```
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6    ?)?Hello!/?t?
0010: d8 f5 43 45             --                            ??CE
```

Run your app from the command line

Throughout this book, you've been running apps from inside the IDE. That's fine for when you're writing an app, but most of the time you want to run it by itself, not in an IDE. To run a console app, you'll typically open a Command or Terminal window and type its filename to run it. Let's do that right now.

Choose View >> Terminal (Ctrl+`) to **open a Terminal window** in the IDE. In Visual Studio, click the **+ Developer Powershell button** to open a prompt. VSCode will bring up the terminal prompt directly. Then run these commands:

```
cd HexDumper/bin/Debug/net8.0
./HexDumper
```

This book uses .NET 8.0—if you're using this with a different version, replace net8.0 with the matching version.

Your app will run, and you'll see the output printed to the console in the terminal.

Use the args variable to access command-line arguments

When you specify command-line arguments, they're passed to the top-level statements using the **`args` variable**, which contains a string array with the command-line arguments. If you're not using top-level statements, the Main method has a `string[] args` parameter.

If the array is empty, there were no arguments passed; otherwise, it contains the arguments.

Here's an example of a simple console app that prints its command-line arguments. **Create a new console app called WriteArgs** and add these top-level statements:

```
Console.WriteLine("I was run with {0} argument(s)", args.Length);
for (int i = 0; i < args.Length; i++)
    Console.WriteLine("Argument #{0}: {1}", i, args[i]);
```

Run your app in the IDE. You'll see it print this line:

```
I was run with 0 argument(s)
```

Now open a Terminal window and run your app from the command line—but this time, pass it arguments:

```
cd WriteArgs/bin/Debug/net8.0
WriteArgs first-argument arg_#2 "third argument"
```

You can include a space in a command-line argument by putting the entire argument in quotes. On macOS and Linux, you'll need to add ./ to the beginning of the command line.

Your app will run, and you'll see this output written to the console in the terminal:

```
I was run with 3 argument(s)
Argument #0: first-argument
Argument #1: arg_#2
Argument #2: third argument
```

When your app runs with command-line arguments, they're populated, and they're passed to the top-level statements or Main method as a string array called args.

Pass command-line arguments to an app run in the IDE

For Visual Studio: you can specify the command-line arguments by choosing WriteArgs Debug Properties from the Debug menu (the menu option will include the project name). Enter each argument in the **Command line arguments box**, one argument per line:

Command line arguments

Command line arguments to pass to the executable. You may break arguments into multiple lines.

```
first-argument
arg_#2
third argument
```

We're using the Debug Properties window to pass a filename to our hex dumper app. For this example, we're using a file that's present on all Windows systems.

This will create a folder called Properties and add a new JSON file called *launchSettings.json* inside of it:

```
{
  "profiles": {
    "WriteArgs": {
      "commandName": "Project",
      "commandLineArgs": "first-argument\r\narg_#2\r\nthird argument"
    }
  }
}
```

This key matches the project name. It will be different if you chose a different name for your project.

The commandLineArgs string has all of the arguments separated by \r\n (escape codes to create a new line in Windows).

For VSCode: At the time we're writing this, VSCode doesn't have a way to automatically generate the *launchProperties.json* file, so you'll need to create it manually. Go to the **Explorer** (***not*** the Solution Explorer) and right-click on the project folder (the second WriteArgs folder, underneath the solution folder that's also called WriteArgs). Choose New Folder (or click) and **create a folder called Properties**. Next, right-click on the new Properties folder and choose Add New File (or click), choose "Custom file (without template)" from the list, and **create a new file called** *launchSettings.json*. Double-click on the new file to open it in the editor. Carefully type in the contents of *launchProperties.json* so they match the file above. If you chose a different name for your project, replace `WriteArgs` on the third line with the name of your project.

Now run your app in the debugger—this time its `args` variable will contain the command-line arguments.

Exercise

Use the `args` variable to make your HexDump app take a command-line argument. If no arguments are passed, or if more than one argument is passed, it writes the following line to the console:

```
usage: HexDump filename
```

If an argument is passed, it uses that argument as a filename. The app opens the file and writes a hex dump of it to the console. Your app will be reading any file, so it might get some weird characters. Modify the code that writes the actual characters in the buffer to replace any byte below 32 with a `'?'` question mark character.

MINI Sharpen your pencil

What happens if you pass the app the name of a file that doesn't exist?

.....

Exercise Solution

Use the `args` variable to make your HexDump app take a command-line argument.

```
if (args.Length != 1)
{
    Console.WriteLine("usage: HexDump filename");
    return;
}

using (var input = File.OpenRead(args[0]))
{
    var position = 0;
    var buffer = new byte[16];
    while (true)
    {
        // Read up to the next 16 bytes from the file into a buffer
        var bytesRead = input.Read(buffer, 0, buffer.Length);
        if (bytesRead == 0) return;

        // Write the position (or offset) in hex, followed by a colon and space
        Console.Write("{0:x4}: ", position);
        position += bytesRead;

        // Write the hex value of each character in the byte array
        for (var i = 0; i < 16; i++)
        {
            if (i < bytesRead)
                Console.Write("{0:x2} ", (byte)buffer[i]);
            else
                Console.Write("   ");
            if (i == 7) Console.Write("-- ");
        }

        // Write the actual characters in the buffer
        for (int i = 0; i < bytesRead; i++)
        {
            if (buffer[i] < 32)
                buffer[i] = (byte)'?';
            Console.Write((char)buffer[i]);
        }
        Console.WriteLine();
    }
}
```

If there isn't exactly one argument passed on the command line, it prints this message.

Instead of reading "textdata.txt" or "binarydata.dat," it opens the file passed on the command line.

Try creating text files with various Unicode characters, then dump them and look for their hex values.

This if statement replaces any byte less than 32 with a question mark. Try taking out that if statement and dumping the contents of HexDump.dll. What happens? Are there extra line breaks in the dump? Do you hear beeps?

We'll learn about how to deal with this problem in Chapter 12. What do you think we should do about it?

MINI Sharpen your pencil Solution

What happens if you pass the app the name of a file that doesn't exist?

The app throws a System.IO.FileNotFoundException with the message "Could not find. file"

We're getting close to the end of the book, so it's time to work on your research skills. Take your time and tackle these three AI research challenges—they'll help you level up your ability to keep learning on your own.

Sens-AI

We've showed you the fundamentals of streaming. Can you use AI to expand your knowledge? See if you can use your favorite AI chatbot to research these topics and write code that works to meet these challenges. Here are three big questions. Use the prompt engineering techniques that we talked about in Chapters 8 and 9 to research each of the questions below, then use that information to write code to meet the challenge.

Why did the console beep?

Try removing the code in your hex dump app that replaces bytes with values under 32, then dump the contents of HexDump.dll or another executable. The output will look a little messy because of the line break characters like '\r' and '\n'—and if you listen carefully you'll hear some beeps too (they may make a pleasant chime sound on Windows, depending on your settings). Here are a few simple prompts to get you started:

- What are control characters?
- Why do some characters beep?
- What is the bell character, and how do I use it in C#?

The Terminal window in VSCode might not beep. You can open a Command or Terminal window and run the app from the command line to hear the beeps. Try setting csharp.debug.console to externalTerminal.

★ *Challenge: Create a console app that beeps in a rhythm. Research the Thread.Sleep method and use it to add a delay between beeps.*

What is ASCII?

One of the first things you'll probably run across in your AI-assisted search for the mysterious beep is ASCII. You'll probably be able to guess from the context that it's some sort of encoding. But what, exactly, is ASCII?

- What is ASCII?
- How is ASCII different from UTF-8?
- What is UTF-7, and how is it different from ASCII?
- How does the byte order mark work, and how does it affect ASCII characters in a file?

When you research the byte order mark, you may learn about "big endian" and "little endian." This is a good research topic to really dig into.

★ *Challenge: Create a console app that writes an ASCII chart to the console. Research "ASCII chart" to see what one looks like. Ask the AI to show you an example of an ASCII chart. Can you make one that looks like a table with eight characters per row? Can you use ASCII line characters to make it look nice?*

What are Standard In, Standard Out, and Standard Error?

Standard input (stdin), standard output (stdout), and standard error (stderr) are standard communication channels that console apps use. You've read from stdin and written to stdout. What is that last one, sterr, for?

- Why would I write to Standard Error instead of Standard Out?
- How do I write to Standard Error in C#?
- How do I run a command so it writes stdout to a file but prints stderr to the console?

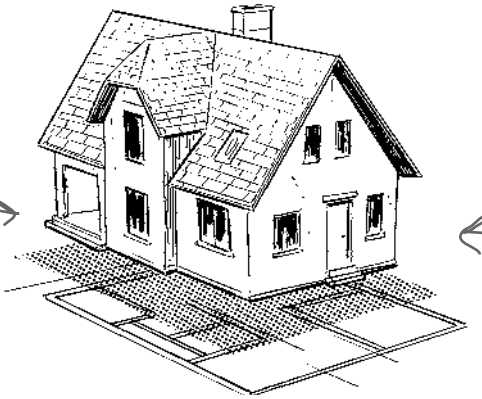
This question might have different answers depending on your OS, so adjust your prompt accordingly.

★ *Challenge: Modify your hex dump app so it prints the "usage" message to stderr instead of stdout. Then run your app and then pipe the output to a file so you can see what's printed to stdout—anything written to the file went to stdout, and anything displayed on the console was written to stderr.*

Downloadable exercise: Hide and Seek

In this next exercise, you'll build an app where you explore a house and play a game of Hide and Seek against a computer player. You'll put your collection and interface skills to the test when you lay out the locations. Then you'll turn it into a game, serializing the state of the game to a file so you can save and load it.

First you'll explore a virtual house, navigating from room to room and examining the items in each location.



Then you'll add a computer player who finds a hiding place. See how few moves it takes to find them!

Go to the GitHub page for the book and download the project PDF:
https://github.com/head-first-csharp/fifth-edition

Bullet Points

- **Unicode** is an industry standard for *encoding* characters, or converting them into bytes. Every one of the over one million Unicode characters has a *code point*, or a unique number assigned to it.
- Most files and web pages are encoded using **UTF-8**, a variable-length Unicode encoding that encodes some characters with either one, two, three, or four bytes.
- C# and .NET use **UTF-16** when storing characters and text in memory, treating a string as a **read-only collection of chars**.
- The **Encoding.UTF8.GetString method** converts a UTF-8 byte array to a string. **Encoding.Unicode** converts a byte array encoded with UTF-16 to a string, and Encoding.UTF32 converts a UTF-32 byte array.
- Use **`\u` escape sequences** to include Unicode in C# strings. The `\u` escape sequence encodes UTF-16, while **`\U`** encodes **UTF-32**, a 4-byte fixed-length encoding.
- StreamWriter and StreamReader work well with text, but will not handle many characters outside of the Latin character sets. Use **BinaryWriter and BinaryReader** to read and write binary data.
- The **StreamReader.ReadBlock** method reads characters into a byte array buffer. It **blocks**, or keeps executing and doesn't return, until it's either read all of the characters you asked for or run out of data to read.
- File.OpenRead returns a FileStream, and the **FileStream.Read method** reads bytes from a stream.
- The String class has an overloaded constructor that takes a char array and converts it to a string. It will convert the entire array to the string, unless it contains a **`\0` null character**, in which case it will end the string when it reaches the null character.
- C# makes the command-line arguments for a console app available as the **`args` variable** that's available to top-level statements (or passed into the Main method).

Unity Lab #5

Raycasting

When you set up a scene in Unity, you're creating a virtual 3D world for the characters in your game to move around in. But in most games, most things in the game aren't directly controlled by the player. So how do these objects find their way around a scene?

The goal of labs 5 and 6 is to get you familiar with Unity's **pathfinding and navigation system**, a sophisticated AI system that lets you create characters that can find their way around the worlds that you create. In this lab, you'll build a scene out of GameObjects and use navigation to move a character around it.

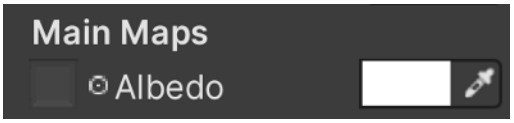
You'll use **raycasting** to write code that's responsive to the geometry of the scene, **capture input**, and use it to move a GameObject to the point where the player clicked. Just as importantly, you'll **get practice writing C# code** with classes, fields, references, and other topics we've discussed.

Create a new Unity project and start to set up the scene

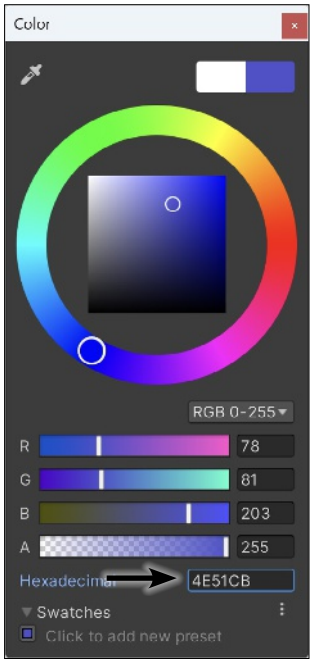
Before you begin, close any Unity project that you have open. Also close your IDE—we'll let Unity open it for us. Create a new Unity project using the 3D template, set your layout to Wide so it matches our screenshots, and give it a name like ***Unity Labs 5 and 6*** *so you can come back to it later.*

Start by creating a play area that the player will navigate around. Right-click inside the Hierarchy window and **create a Plane** (GameObject >>3D Object >> Plane). Name your new Plane GameObject *Floor*.

Right-click on the Assets folder in the Project window and **create a folder inside it called Materials**. Then right-click on the new Materials folder you created and choose **Create >> Material**. Call the new material *FloorMaterial*. Let's keep this material simple for now—we'll just make it a color. Select Floor in the Project window, then click on the white box to the right of the word "Albedo" in the Inspector.

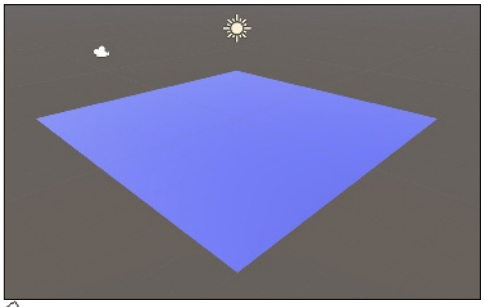


You can use this dropper to grab a color from anywhere on your screen.



In the Color window, use the outer ring to choose a color for the floor. We used a color with number 4E51CB in the screenshot—you can type that into the Hexadecimal box.

Drag the material from the **Project window onto the Plane GameObject in the Hierarchy window**. Your floor plane should now be the color that you selected.



Brain Power

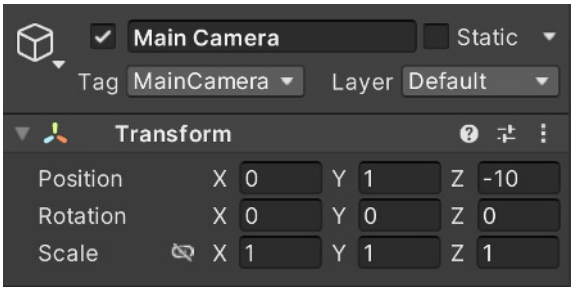
A Plane has no Y dimension. What happens if you give it a large Y scale value? What if the Y scale value is negative? What if it's zero?

Think about it and take a guess. Then use the Inspector window to try various Y scale values and see if the plane acts the way you expected. (Don't forget to set them back!)

A *Plane* is a flat square object that's 10 units long by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane). Unity creates it so that the center of the plane is at point (0,0,0). This center point of the plane determines its position in the scene. Just like our other objects, you can move a plane around the scene by using the Inspector or the tools to change its position and rotation. You can also change its scale, but since it has no height, you can only change the X and Z scale—any positive number you put into the Y scale will be ignored.
The objects that you can create using the 3D Object menu (planes, spheres, cubes, cylinders, and a few other basic shapes) are called *primitive objects*. You can learn more about them by opening the Unity Manual from the Help menu and searching for the "*Primitive and placeholder objects*" help page. Take a minute and open up that help page right now. Read what it says about planes, spheres, cubes, and cylinders.

Set up the camera

In the past two Unity Labs, you learned that a GameObject is essentially a "container" for components, and that the Main Camera has just three components: a Transform, a Camera, and an Audio Listener. That makes sense, because all a camera really needs to do is be at a location and record what it sees and hears. Have a look at the camera's Transform component in the Inspector window.



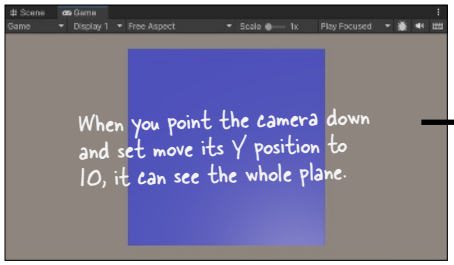
Notice how the position is (0, 1, –10). Click on the Z label in the Position line and drag up and down. You'll see the camera fly back and forth in the scene window. Take a close look at the box and four lines in front of the camera. They represent the camera's **viewport**, or the visible area on the player's screen.

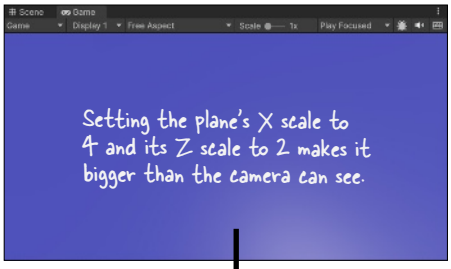
Move the camera around the scene and rotate it using the Move tool (W) and Rotate tool (E), just like you did with other GameObjects in your scene.

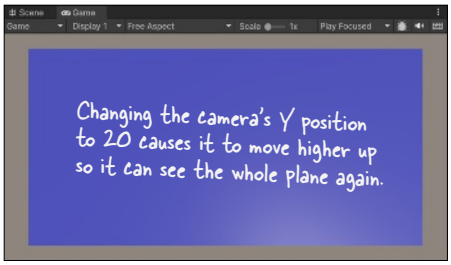
Use the context menu in the Inspector window to reset the Main Camera's Transform component. Notice how it ***doesn't reset the camera to its original position***—it resets both the camera's position and its rotation to (0, 0, 0). You'll see the camera intersecting the plane in the Scene window.

Now let's point the camera straight down. Start by clicking on the X label next to Rotation and dragging up and down. You'll see the viewport move. Now **set the camera's X rotation to 90** in the Inspector window to point it straight down.

Switch to the Game panel by clicking the Game tab or choosing *Panels >> Game* from the Window menu to see what the main camera sees. You won't see anything—which makes sense, because the camera is looking straight down below the infinitely thin plane. **Click on the Y position label in the Transform component and drag up** until the Y value is 10 and the camera can see the entire plane.







Switch between the Scene and Game panels to see what the camera sees.

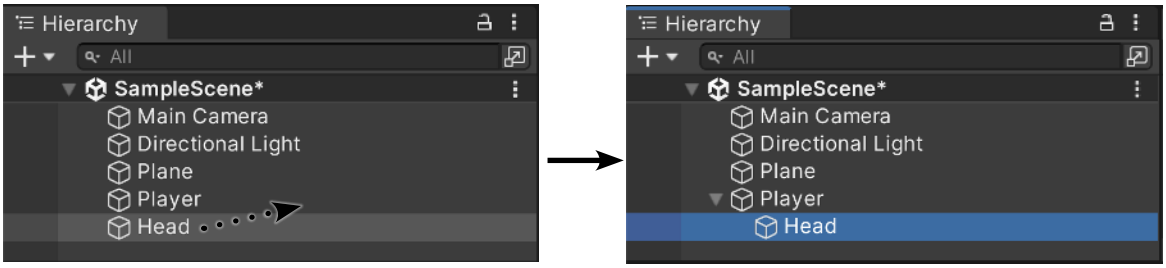
Now **select Floor in the Hierarchy window**. Use the Plane's Transform component in the Inspector window to **set the Floor GameObject's scale to (4, 1, 2)** so that it's twice as long as it is wide. Since a Plane is 10 units wide and 10 units long, this scale will make it 40 units long and 20 units wide. The plane will completely fill up the viewport again, so move the Camera further up along the Y axis until its Y value is 20 and the entire expanded plane is in view again.

Create a GameObject for the player

Your game will need a player to control. We'll create a simple humanoid-ish player that has a cylinder for a body and a sphere for a head. Click the Scene tab or choose Panels >> 5 Scene from the Window menu to go back to the Scene view. Make sure you don't have any objects selected by clicking the scene (or the empty space) in the Hierarchy window.

Create a Cylinder GameObject (3D Object >> Cylinder)—you'll see a cylinder appear in the middle of the scene. Change its name to *Player*, then **choose Reset from the context menu** for the Transform component to make sure it has all of its default values. Next, **create a Sphere GameObject** (3D Object >> Sphere). Change its name to *Head*, and reset its Transform component as well. They'll each have a separate line in the Hierarchy window.

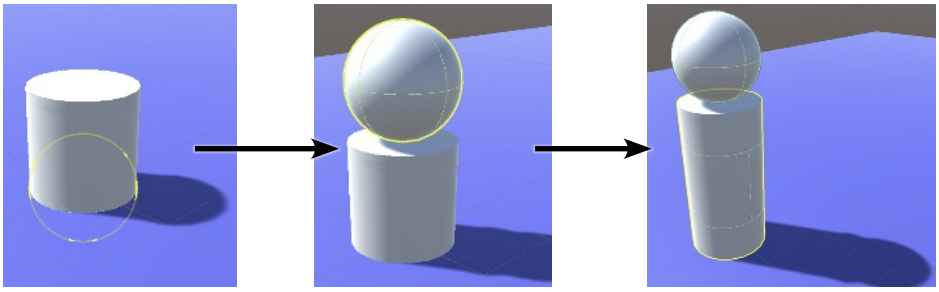
But we don't want separate GameObjects—we want a single GameObject that's controlled by a single C# script. This is why Unity has the concept of **parenting**. Click on Head in the Hierarchy window and **drag it onto Player**. This makes Player the parent of Head. Now the Head GameObject is **nested** under Player.



Select Head in the Hierarchy window. It was created at (0, 0, 0) like all of the other spheres you created. You can see the outline of the sphere, but you can't see the sphere itself because it's hidden by the plane and the cylinder. Use the Transform component in the Inspector window to **change the Y position of the sphere to 1.5**. Now the sphere appears above the cylinder, just the right place for the player's head.

Now select Player in the Hierarchy window. Since its Y position is 0, half of the cylinder is hidden by the plane. **Set its Y position to 1.** The cylinder pops up above the plane. Notice how it took the Head sphere along with it. Moving Player causes Head to move along with it because moving a parent GameObject moves its children too—in fact, *any* change that you make to its Transform component will automatically get applied to the children. If you scale it down, its children will scale too.

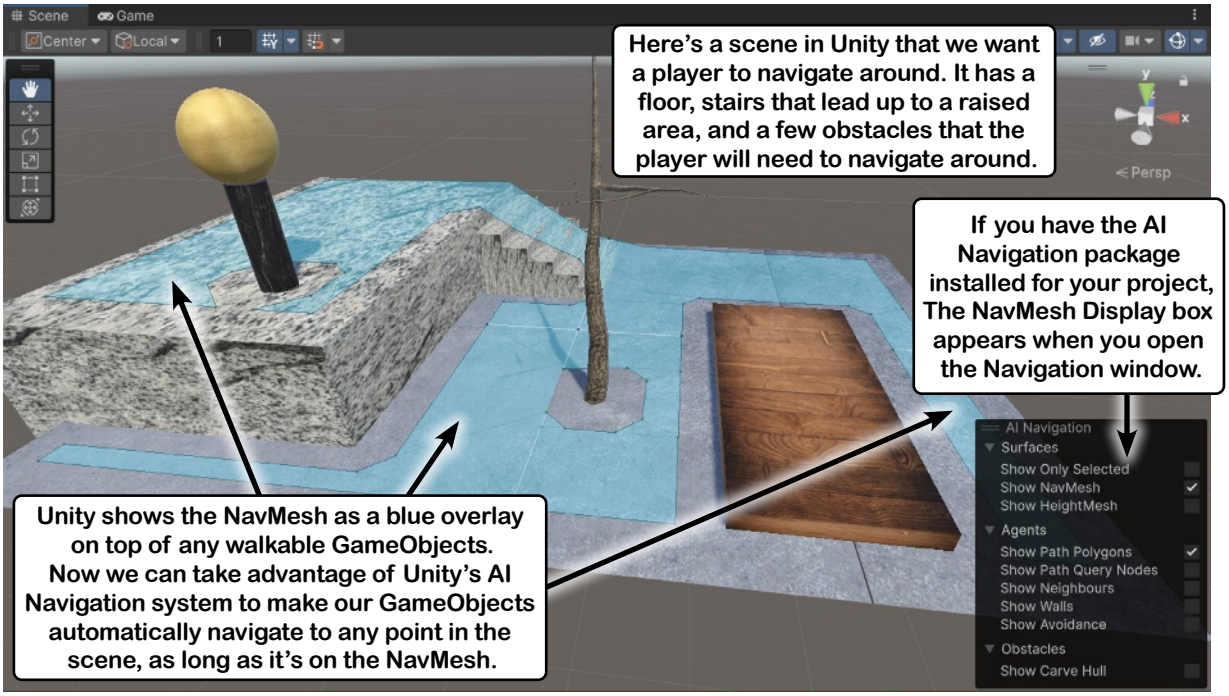
Switch to the Game window—your player is in the middle of the play area.



When you modify the Transform component for a GameObject that has nested children, the children will move, rotate, and scale along with it.

Introducing Unity's navigation system

One of the most basic things that video games do is move things around. Players, enemies, characters, items, obstacles… all of these things can move. That's why Unity is equipped with a sophisticated artificial intelligence–based navigation and pathfinding system to help GameObjects move around your scenes. We'll take advantage of the navigation system to make our player move toward a target. You'll start by adding a **navigation mesh** or **NavMesh**, which adds a map of all of the "walkable" places in your scene—all of the places where the navigation system can move GameObjects.



Here's a scene in Unity that we want a player to navigate around. It has a floor, stairs that lead up to a raised area, and a few obstacles that the player will need to navigate around.

If you have the AI Navigation package installed for your project, The NavMesh Display box appears when you open the Navigation window.

Unity shows the NavMesh as a blue overlay on top of any walkable GameObjects. Now we can take advantage of Unity's AI Navigation system to make our GameObjects automatically navigate to any point in the scene, as long as it's on the NavMesh.

Unity provides a sophisticated AI navigation and pathfinding system that can move your GameObjects around a scene in real time by finding an efficient path that avoids obstacles.

Brain Power

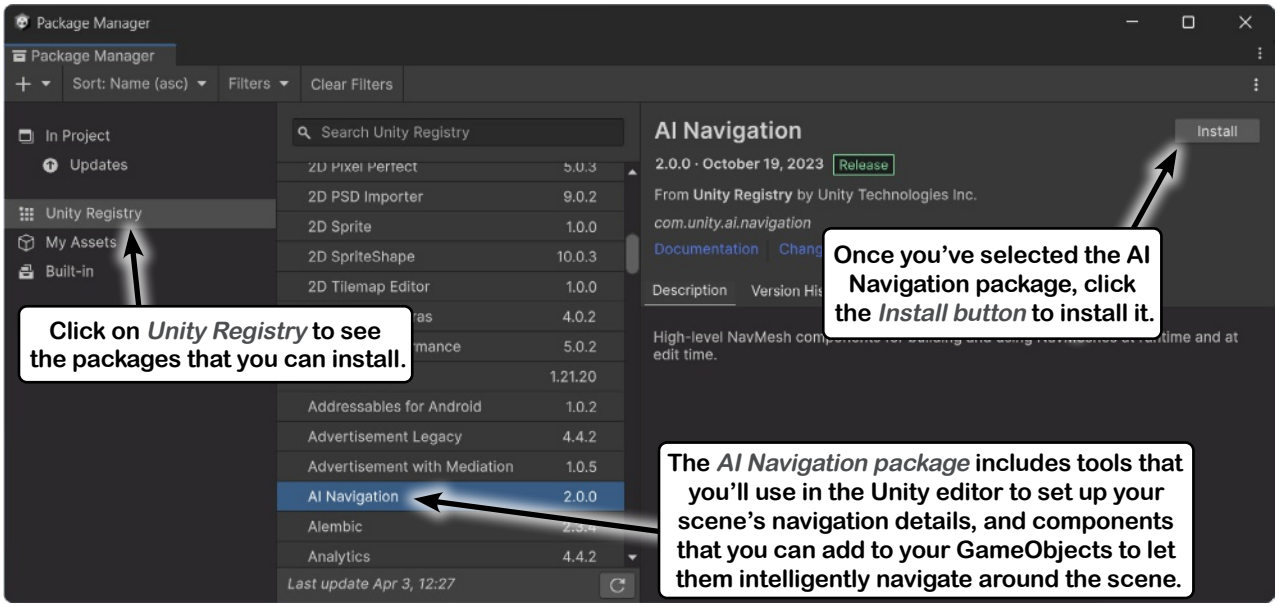
Take a look at the scene in the previous screenshot. The NavMesh shows you all of the areas that a GameObject can navigate to. If you look closely, you'll see that it goes up the steps, and that it doesn't include the tree in the middle of the scene or the sculpture on the upper level.

How would you write a C# script to make a GameObject automatically move from one point on the NavMesh to another, finding the shortest route and never leaving the NavMesh?

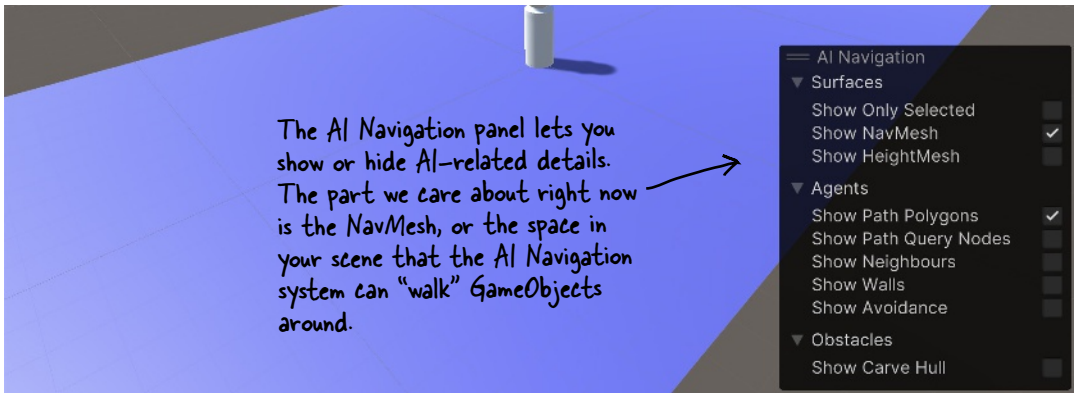
Install the AI Navigation package

The first step in using Unity's AI Navigation system is to **install the AI Navigation package**, which installs navigation components that you can add to your GameObjects and tools that you'll use in the Unity editor.

1. **Choose Package Manager** from the Window menu.
2. Click **Unity Registry** to see the packages that you can install.
3. Select the **AI Navigation** package (you can scroll down to it or enter it in the search box) to display the details of the package in the right side of the Package Manager window.
4. **Click the Install button** to install it.



Once it's installed, you'll see an **AI Navigation panel** at the lower right corner of your scene.



Things you'll do with navigation

Now that you have the AI Navigation package installed, you can use Unity's navigation and pathfinding system to make your characters intelligently find their way around a game world. To use it, you need to set up basic pieces to tell Unity where the player can go:

- First, you need to tell Unity exactly where your characters are allowed to go. You do this by **setting up a NavMesh**, which contains all of the information about the walkable areas in the scene: slopes, stairs, obstacles, and even points called off-mesh links that let you set up specific player actions like opening a door.
- Second, you **add a NavMesh Agent component** to any GameObject that needs to navigate. This component automatically moves the GameObject around the scene, using its AI to find the most efficient path to a target and avoiding obstacles and, optionally, other NavMesh Agents.
- It can sometimes take a lot of computation for Unity to navigate complex NavMeshes. Navigation meshes can get very complex, and all the extra computing power required to navigate them can have a huge impact on the frame rate of a game. Luckily, Unity does this for you—the AI Navigation package has a Bake feature, which lets you set up a NavMesh in advance **and precompute (or bake)** the geometric details to make the agents work more efficiently.

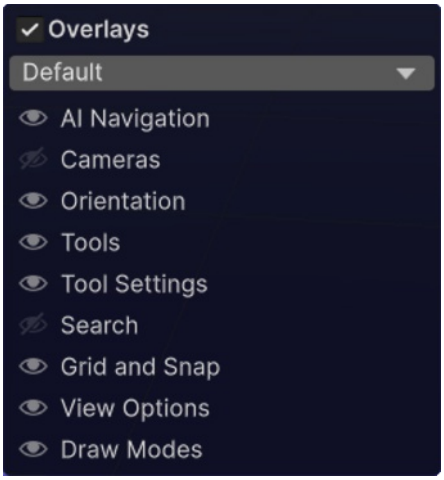
Once you use Unity's AI Navigation tools to set up a NavMesh for your scene, you can add a NavMesh Agent component to a GameObject to move it around the scene automatically.

Watch it!

If you hide the AI Navigation panel, you can use the Overlay Menu in the Scene window to show it again—and to see camera previews.

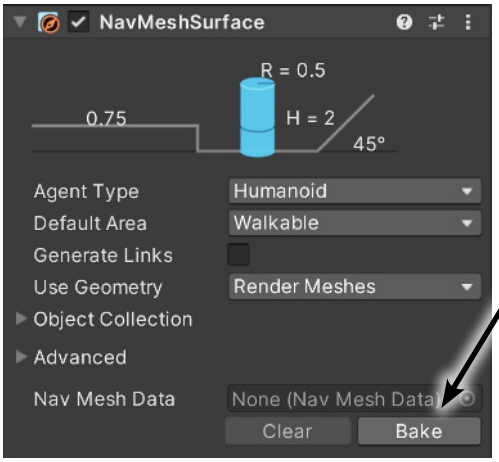
You can hide the AI Navigation panel by right-clicking on the panel and choosing Hide, but then it disappears—and it's not immediately obvious how to get it back!

*The AI Navigation panel is an **overlay**, or a floating panel that you can move around the Scene window (just lke the Tools overlay that you've been using). You can show or hide overlays with the **Overlays Menu**. Click the scene context menu icon ( ) in the upper-right corner of the Scene window, then choose Overlay Menu to display it. Click the show/hide icon ( ) next to the AI Navigation overlay to show or hide it. You can do the same with the other ovelays. Try turning on the **Cameras** overlay—it displays a window that shows what your camera sees so you don't need to switch to the Game panel to see it.*



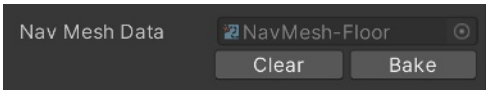
Set up the NavMesh

Let's set up a NavMesh that just consists of the Floor plane. **Add a NavMeshSurface component** to your Floor plane GameObject. Go to the component in the Inspector and **click the Bake button**.



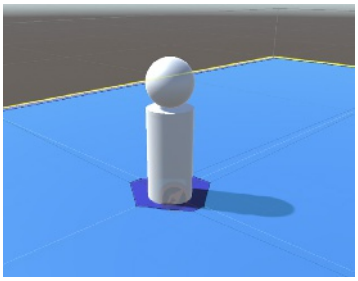
When you click the *Bake button* at the bottom of the NavMeshSurface component in the Inspector, the AI Navigation system looks for GameObjects in the scene with a NavMeshSurface component and *bakes* (or precalculates) a NavMesh that covers every walkable surface of those GameObjects.

Once the NavMesh is baked, the NavMesh Data property will change from None to something else and the Clear button will be enabled. You can recalculate the NavMesh by clicking the Bake button again.

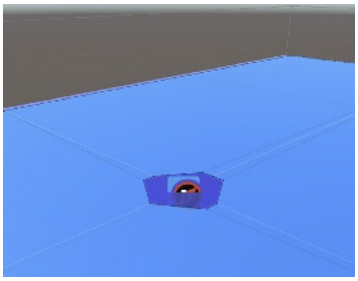


Fix the carved-out hole in your NavMesh

Look closely at the NavMesh—specifically, the floor near the Player GameObject. When you bake the NavMesh, the AI Navigation system **carves holes for any obstacles**. The Player cylinder intersects with the Floor plane, so the navigation system treats it like an obstacle and carves a hole in the NavMesh so any GameObject can navigate around it. To fix this, you'll turn the Player into a prefab. Then you'll delete it from the scene, and click the Bake button to bake the NavMesh again—this time without a hole.



When you baked the NavMesh, it carved a hole around the Player GameObject because its cylinder intersected the Floor plane. You'll get rid of the hole by removing the Player (saving it as a prefab), re-baking the NavMesh, and adding the Player back to the scene.



1. Create a Prefabs folder and **drag the Player GameObject into** it to turn it into a prefab. Since you nested the Head sphere under the Player cylinder, it will be part of the prefab too.
2. **Delete the Player GameObject** from the scene. The nested Head will be deleted too.
3. Go back to the NavMeshSurface component in the Floor and **click the Bake button** again.
4. Zoom into the middle of the plane in the Scene view. The hole should be gone from the NavMesh.

Exercise

You're going to be moving the player around a scene in Unity Labs 5 and 6, and it will be a lot easier for you to see what's going on if you can see it from different angles. We'll do this by making the camera always point at the player:

- Create a Scripts folder and **add a script called MoveCamera** with a Start method that instantiates the player.
- The script will **keep the player in the center of the view** by rotating the camera so it always "looks" at the player.
- The player can **use the cursor keys to rotate the camera** and the **mouse scroll wheel to zoom in and out.**

Instantiate the player prefab in the Start method

We learned about attributes in Chapter 9. You used the [TestClass] and [TestMethod] attributes to write unit tests.

Start by creating a script called MoveCamera. Give it the following fields:

```
[SerializeField]
private GameObject player;

[SerializeField]
private float angle = 1F;

[SerializeField]
private float zoomSpeed = 0.25F;

private GameObject playerInstance;
```

In Chapter 10, we learned about serialization, where your objects are converted to data that can be written to files. Unity uses its own kind of serialization to save and set fields in your scripts: when you change the value of a field in the Inspector, Unity serializes that value. Unity automatically serializes public fields, but making all of your fields public is poor encapsulation. You can add the *SerializeField attribute* to a private field to tell Unity to display it in the Inspector and serialize its value too.

In the Start method, use the Instantiate method and the `player` field to instantiate the player prefab, saving the reference to the instantiated GameObject in the PlayerInstance field. The **transform.LookAt method** makes a GameObject look at a specific position. Add this line to the Start method to make the camera look at the player:

```
transform.LookAt(playerInstance.transform);
```

Modify the Update method to rotate the camera around the player when the arrow keys are pressed

In the Update method, you can check if the player is currently holding down a key by calling the Input.GetKey method. It takes one parameter, a KeyCode enum value like KeyCode.RightArrow, and returns true if that key is currently being pressed. If the player is holding down the right arrow key, use the Main Camera's transform.RotateAround method to rotate the camera by calling this method, passing it a vector to tell it the axis to rotate around:

```
private void RotateCamera(Vector3 axis) =>
    transform.RotateAround(playerInstance.transform.position, axis, angle);
```

If the player is holding the right arrow, rotate around the Vector3.up axis. If they're pressing the left arrow, rotate around Vector3.down. For the up arrow, rotate around Vector3.right, and for the down arrow, rotate around Vector3.left. Then check if the camera's Y position is greater than 1—if it is, set it to a new vector with 1 as the Y value and the current X and Z values. This will keep the camera from going underneath the floor.

Make the scroll wheel zoom the camera in and out and look at the player again

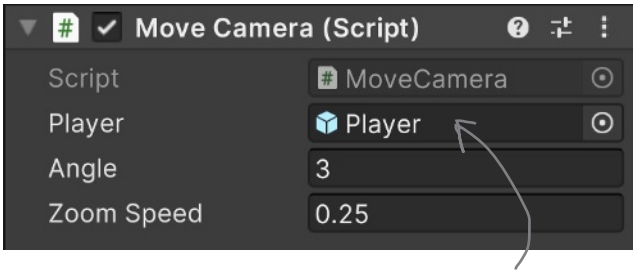
Add this code to the Update method to move the camera closer to or further from the player:

```
var scrollWheelValue = Input.GetAxis("Mouse ScrollWheel");
if (scrollWheelValue != 0)
    transform.position *= (1F + scrollWheelValue * zoomSpeed);
```

Then add one more line at the end of the Update method to make the camera look at the player again.

Exercise Solution

Adding the [SerializeField] attribute to the private fields causes them to show up in the Inspector. Make sure you set the Player field to the prefab that you created earlier.



Don't forget to set the Player field to the prefab!

```
public class MoveCamera : MonoBehaviour
{
    [SerializeField]
    private GameObject player;

    [SerializeField]
    private float angle = 1F;

    [SerializeField]
    private float zoomSpeed = 0.25F;

    private GameObject playerInstance;

    void Start()
    {
        // Instantiate the player prefab and save the reference
        playerInstance = Instantiate(player);

        // Make the camera look at the player.
        transform.LookAt(playerInstance.transform);
    }

    void Update()
    {
        // Rotate the camera around the player when arrow keys are pressed
        if (Input.GetKey(KeyCode.RightArrow))
            RotateCamera(Vector3.up);
        else if (Input.GetKey(KeyCode.LeftArrow))
            RotateCamera(Vector3.down);
        else if (Input.GetKey(KeyCode.UpArrow))
            RotateCamera(Vector3.right);
        else if (Input.GetKey(KeyCode.DownArrow))
            RotateCamera(Vector3.left);

        // Keep the camera above the floor
        if (transform.position.y < 1)
            transform.position = new Vector3(
                                    transform.position.x, 1, transform.position.z);

        // Zoom the camera in and out with the mouse scroll wheel.
        var scrollWheelValue = Input.GetAxis("Mouse ScrollWheel");
        if (scrollWheelValue != 0)
            transform.position *= (1F + scrollWheelValue * zoomSpeed);

        // Make the camera look at the player again
        transform.LookAt(playerInstance.transform);
    }

    private void RotateCamera(Vector3 axis) =>
        transform.RotateAround(playerInstance.transform.position, axis, angle);
}
```

The transform.LookAt method causes a GameObject to rotate to point to another GameObject—in this case, it turns the camera to point at the player.

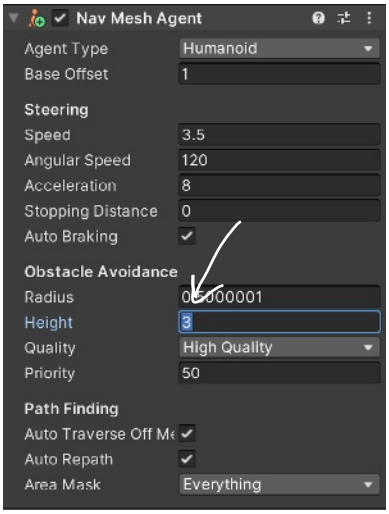
Input.GetAxis("Mouse ScrollWheel") returns a number (usually between –0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move). We used vector arithmetic to zoom the camera by multiplying transform.position by (1F + scrollWheelValue * ZoomSpeed).

transform.position is a vector, so multiplying it with *= operator causes it to shrink or grow, which means it moves toward or away from (0, 0, 0).

If your camera isn't rotating, make sure you set the Player field in the Move Camera script component.

Make your player automatically navigate the play area

Add a NavMesh Agent to your Player GameObject. **Select Player** in the Hierarchy window, then go back to the Inspector window, click the **Add Component** button, and choose **Navigation >> Nav Mesh Agent** to add the NavMesh Agent component. The cylinder body is 2 units tall and the sphere head is 1 unit tall, so you want your agent to be 3 units tall—so **set the Height to 3**. Now the NavMesh Agent is ready to move the Player GameObject around the NavMesh.



Add a script called *MoveToClick.cs* to the Scripts folder you created in the exercise. This script will let you click on the play area and tells the NavMesh Agent to move the GameObject to that spot. You learned about private fields in Chapter 5. This script will use one to store a reference to the NavMeshAgent. Your GameObject's code will need a reference to its agent so it can tell the agent where to go, so you'll call the GetComponent method to get that reference and save it in a **private NavMeshAgent field** called `agent`:

```
agent = GetComponent<NavMeshAgent>();
```

Here's the **code for your MoveToClick script** (the navigation system uses classes in the UnityEngine.AI namespace, so include the `using` statement):

```
using UnityEngine.AI;

public class MoveToClick : MonoBehaviour
{
    private NavMeshAgent agent;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
            Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100))
            {
                agent.SetDestination(hit.point);
            }
        }
    }
}
```

In the last Unity Lab, you used the Start method to set a GameObject's position when it first appears. There's actually a method that gets called before your script's Start method. The *Awake* method is called when the object is created, while Start is called when the script is enabled. The MoveToClick script uses the Awake method to initialize the field, not the Start method.

Here's where the script handles *mouse clicks*. The Input.GetMouseButtonDown method checks if the user is currently pressing a mouse button, and the 0 argument tells it to check for the left button. Since Update is called every frame, it's always checking to see if the mouse button is clicked.

Experiment with the Speed, Angular Speed, Acceleration, and Stopping Distance fields in the NavMesh agent. You can change them while the game is running (but remember it won't save any values that you change while running the game). What happens when you make some of them really big?

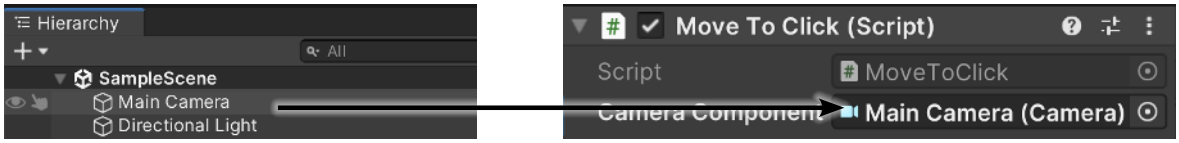
Add the script to the Player *prefab* and run the game. While the game is running, **click anywhere on the floor**. When you click on the plane, the NavMesh Agent will move your player to the point that you clicked.

Sharpen your pencil

We've talked a lot about object references and reference variables over the last few chapters. Let's do a little pencil-and-paper work to get some of the ideas and concepts behind object references into your brain. First, add this public field to the MoveToClick class:

```
public Camera cameraComponent;
```

Go back to the Hierarchy window, click on Player, and find the new Main Camera field in the Move To Click (Script) component. Then **drag the Main Camera** out of the Hierarchy window and **onto the Camera Component field** in the Player GameObject's Move To Click (Script) component in the inspector:



Now **comment out** this line:

```
// Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
```

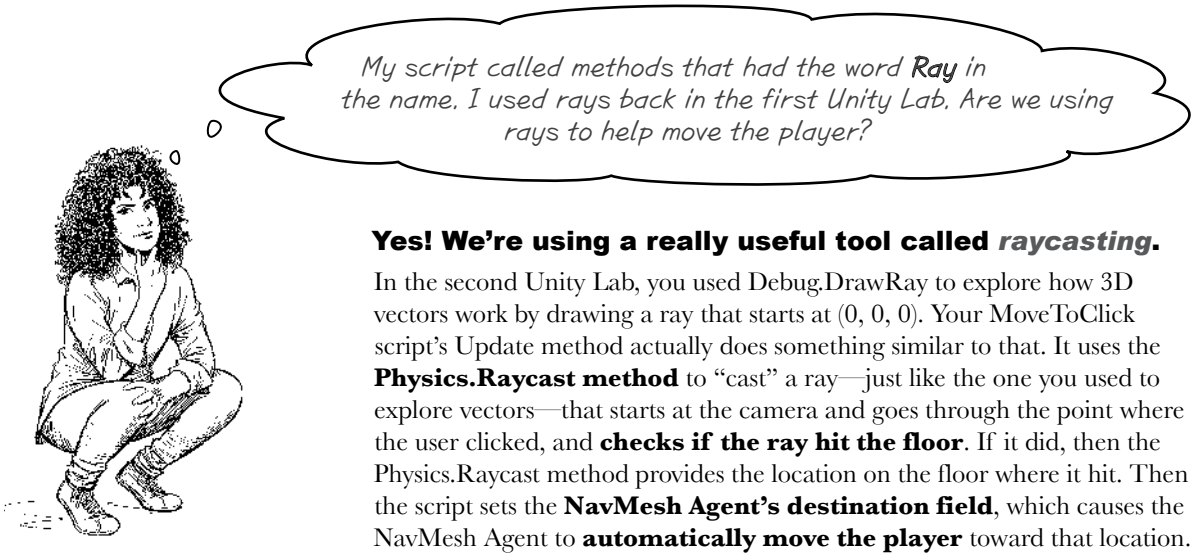
Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

.....

.....

.....

.....



Yes! We're using a really useful tool called *raycasting*.

In the second Unity Lab, you used Debug.DrawRay to explore how 3D vectors work by drawing a ray that starts at (0, 0, 0). Your MoveToClick script's Update method actually does something similar to that. It uses the **Physics.Raycast method** to "cast" a ray—just like the one you used to explore vectors—that starts at the camera and goes through the point where the user clicked, and **checks if the ray hit the floor**. If it did, then the Physics.Raycast method provides the location on the floor where it hit. Then the script sets the **NavMesh Agent's destination field**, which causes the NavMesh Agent to **automatically move the player** toward that location.

Raycasting Up Close

Your MoveToClick script calls the **Physics.Raycast method**, a really useful tool Unity provides to help your game respond to changes in the scene. It shoots (or "casts") a virtual ray across your scene and tells you if it hit anything. The Physics.Raycast method's parameters tell it where to shoot the ray and the maximum distance to shoot it:

`Physics.Raycast(`where to shoot the ray`, out hit,` maximum distance`)`

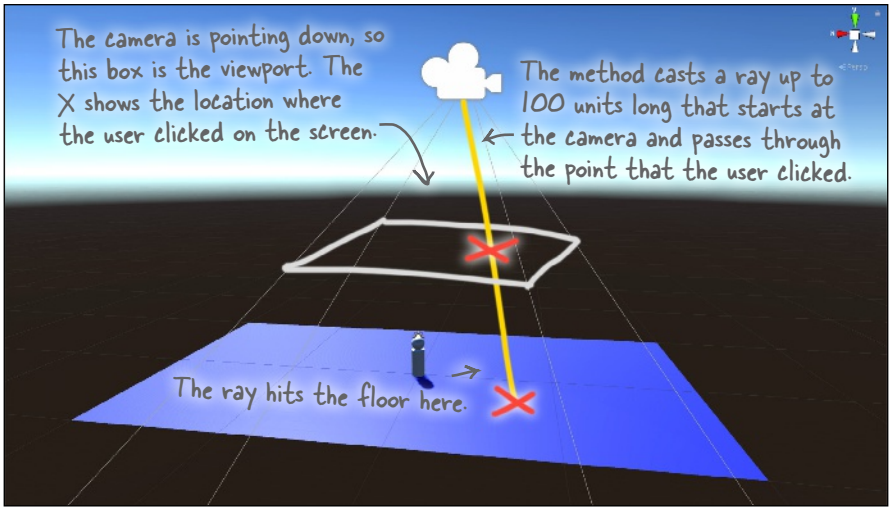
This method returns true if the ray hit something, or false if it didn't. It uses the `out` keyword to save the results in a variable, exactly like you saw with int.TryParse in the last few chapters. Let's take a closer look at how this works.

We need to tell Physics.Raycast where to shoot the ray. So the first thing we need to do is find the camera—specifically, the Camera component of the Main Camera GameObject. Your code gets it just like you got the GameController in the last Unity Lab:

```
GameObject.Find("Main Camera").GetComponent<Camera>();
```

The Camera class has a method called ScreenPointToRay that creates a ray that shoots from the camera's position *through an (X, Y) position on the screen*. The Input.mousePosition method provides the (X, Y) position on the screen where the user clicked. This ray provides you with the location to feed into Physics.RayCast:

```
Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
```



Now that the method has a ray to cast, it can call the Physics.Raycast method to see where it hits:

```
RaycastHit hit;
if (Physics.Raycast(ray, out hit, 100))
{
    agent.SetDestination(hit.point);
}
```

It returns a bool and uses the `out` keyword—in fact, it works exactly like int.TryParse. If it returns true, then the `hit` variable contains the location on the floor that the ray hit. Setting agent.destination tells the NavMesh Agent to start moving the player toward the point where the ray hit.

Sharpen your pencil Solution

We gave you a pencil-and-paper exercise to do. You modified the MoveToClick class to add a field for the Main Camera instead of using the Find and GetComponent methods. We had you drag the Main Camera onto it, then we asked you a question. Was your answer similar to ours?

Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

When my code called mainCamera.GetComponent<Camera>, it returned a reference to a GameObject. I replaced it with a field and dragged the Main Camera GameObject from the Hierarchy window into the Inspector window, which caused the field to be set to a reference to the same GameObject. Those were two different ways to set the cameraComponent variable to reference the same object, which is why it behaved the same way.

You'll be reusing the MoveToClick script in later Unity Labs, so after you're done writing down the answers, *change the script back to the way it was* by removing the MainCamera field and restoring the line that sets the `cameraComponent` variable.

Bullet Points

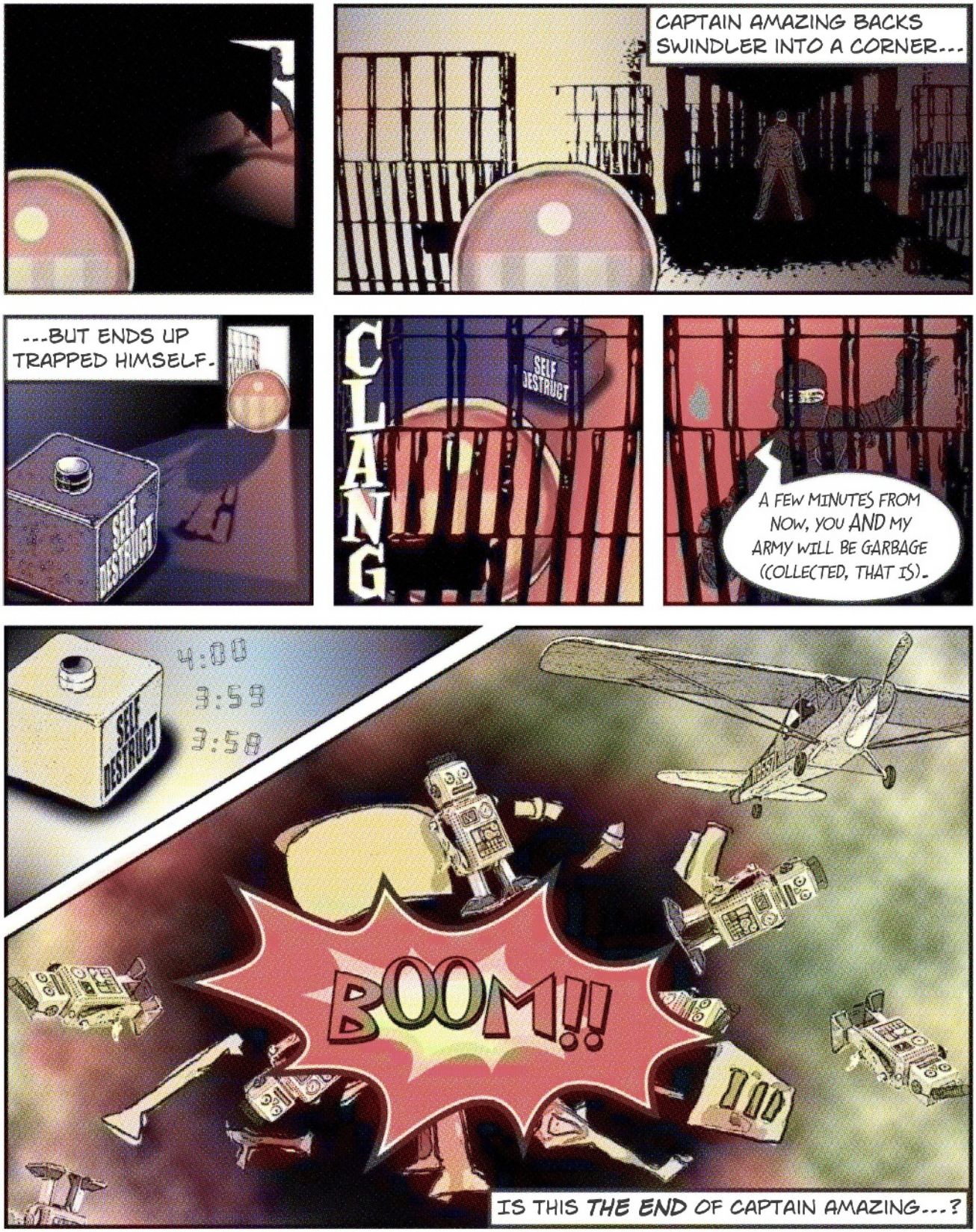
- A **Plane** is a flat square object that's 10 units by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane).
- You can **move the Main Camera** to change the part of the scene that it captures by modifying its Transform component, just like you move any other GameObject.
- When you modify the Transform component of a GameObject that has **nested children**, the children will move, rotate, and scale along with it.
- Unity's **AI navigation and pathfinding system** can move your GameObjects around a scene in real time by finding an efficient path that avoids obstacles.
- A **NavMesh** contains all of the information about the walkable areas in the scene. Set up a NavMesh in advance and precompute—or bake—its details to make the agents work more efficiently. A **NavMesh Surface component** includes GameObject in the NavMesh.
- A **NavMesh Agent** component automatically moves a GameObject around the scene, using its AI to find the most efficient path to a target.
- The **NavMeshAgent.SetDestination method** triggers the agent to calculate a path to a new position and start moving toward the new destination.
- Unity calls your script's **Awake method** when it first loads the GameObject, well before it calls the script's Start method but after it instantiates other GameObjects.
- The **Input.GetKey** method returns true if a key is down.
- **Input.GetAxis** returns a float value that corresponds to mouse or joystick movement. Passing it the string "Mouse ScrollWheel" causes it to check the scroll wheel.
- The **Input.GetMouseButtonDown method** returns true if a mouse button is currently being clicked.
- The **Physics.Raycast** method does *raycasting* by shooting (or "casting") a virtual ray across the scene and returns true if it hit anything. It uses the `out` keyword to return information about what it hit.
- The camera's **ScreenPointToRay** method creates a ray that goes through a point on the screen. Combine it with Physics.Raycast to determine where to move the player.

CAPTAIN AMAZING

THE DEATH OF THE OBJECT

Head First C#	
Four bucks	Chapter 11

CAPTAIN AMAZING, OBJECTVILLE'S MOST AMAZING OBJECT, PURSUES HIS ARCHNEMESIS...
I'VE GOT YOU NOW, SWINDLER.
YOU'RE TOO LATE! AS WE SPEAK MY CLONE ARMY IS GATHERING IN THE FACTORY BENEATH US...
...READY TO WREAK HAVOC ON THE STREETS OF OBJECTVILLE!
Welcome to
Objectville
Home of Polymorphism
POW!!
I'LL TAKE DOWN EACH CLONE'S REFERENCES, ONE BY ONE.

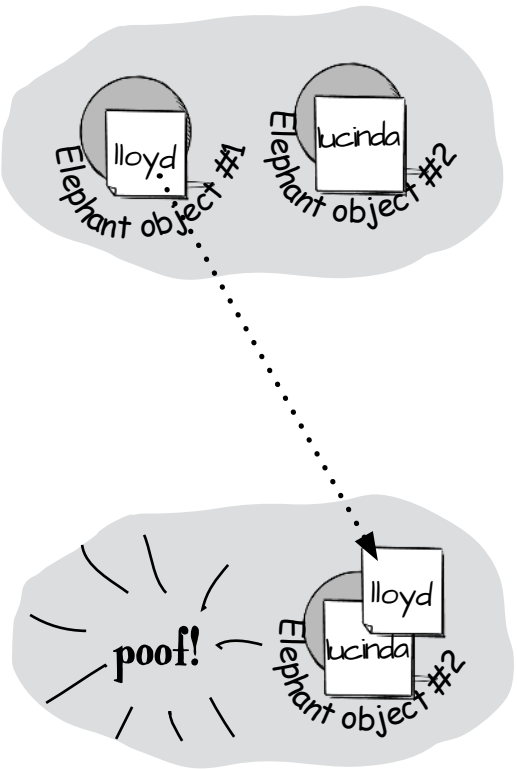

CAPTAIN AMAZING BACKS SWINDLER INTO A CORNER...
...BUT ENDS UP TRAPPED HIMSELF.
SELF DESTRUCT
CLANG
SELF DESTRUCT
A FEW MINUTES FROM NOW, YOU AND MY ARMY WILL BE GARBAGE (COLLECTED, THAT IS).
4:00
3:59
3:58
SELF DESTRUCT
BOOM!!
IS THIS THE END OF CAPTAIN AMAZING...?

The life and death of an object

Here's a quick review of what we know about how objects live and die:

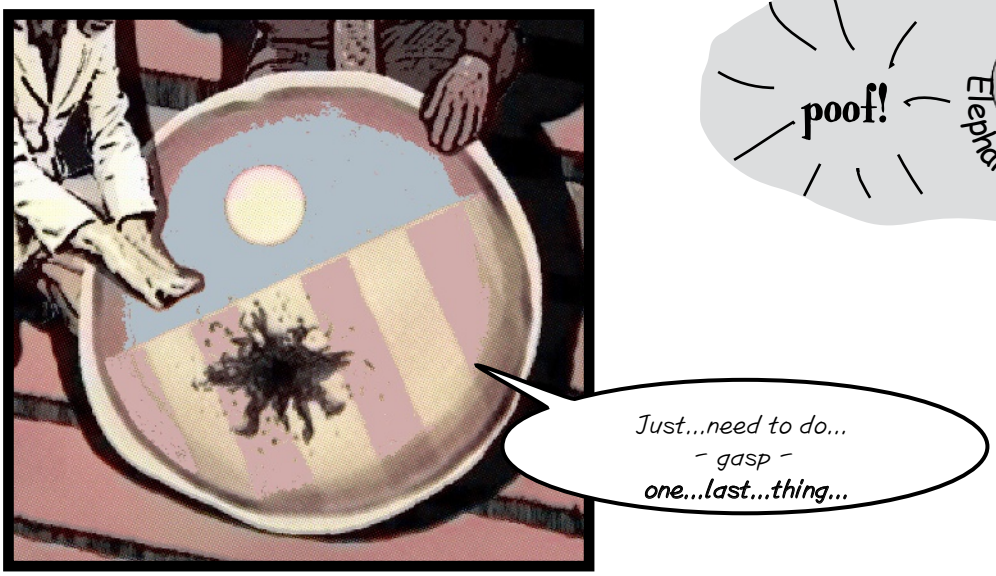
- When you create an object, the CLR—which runs your .NET applications and manages memory—allocates enough memory for it on the heap, a special portion of your computer's memory reserved for objects and their data.
- It's kept "alive" by a reference, which can be stored in a variable, a collection, or a property or field of another object.
- There can be lots of references to the same object—like you saw in Chapter 4, when you pointed the `lloyd` and `lucinda` reference variables to the same instance of Elephant.
- When you took away the last reference to the Elephant object, that caused the CLR to mark it for garbage collection.
- Eventually the CLR removed the Elephant object and the memory was reclaimed so it could be used for new instances of objects that your program would go on to create later.

Here's a picture from back in Chapter 4. You created two Elephant objects on the heap, then you removed the reference to one of them to mark it for garbage collection. But what does that actually mean? What's doing the collecting?



Now we'll explore all of these points in more detail, writing small programs that help show how garbage collection works.

But before we can start experimenting with garbage collection, we need to take a step back. You learned earlier that objects are "marked" for garbage collection—but that the actual removal of the object can happen at any time (or never!). We'll need a way to know when an object has been garbage-collected, and a way to force that garbage collection to happen. So that's where we'll start.



Use the GC class (with caution) to force garbage collection

.NET gives you a **GC class** that controls the garbage collector. We'll use its static methods—like GetTotalMemory, which returns a long with an *approximate* count of the number of bytes currently *thought* to be allocated on the heap:

```
Console.WriteLine(GC.GetTotalMemory(false));
```

class System.GC
Controls the system garbage collector, a service that automatically reclaims unused memory.

You may be thinking, "Why *approximate*? What does *thought* to be allocated mean? How can the garbage collector not know exactly how much memory is allocated?" That reflects one of the basic rules of garbage collection: you can absolutely, 100% rely on garbage collection, but ***there are a lot of unknowns and approximations***.

In this chapter, we're going to use a few GC functions:

- GC.GetTotalMemory returns the approximate number of bytes currently thought to be allocated on the heap.
- GC.GetTotalAllocatedBytes returns the approximate number of bytes that have been allocated since the program was started.
- GC.Collect forces the garbage collector to reclaim all unreferenced objects immediately.

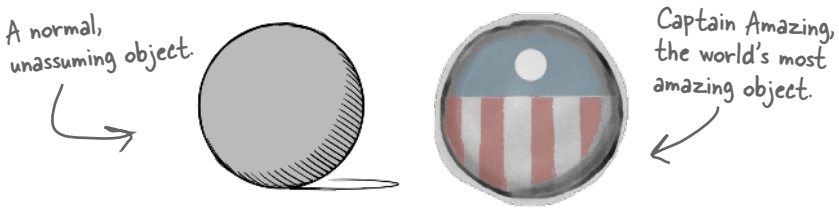
There's just one thing about these methods: we're using them for learning and exploration, but unless you ***really*** know what you're doing, **do not call GC.Collect in code for a real project**. The .NET garbage collector is a finely tuned, carefully calibrated piece of engineering. In general, when it comes to determining when to collect objects, it's smarter than we are, and we should trust it to do its job.

there are no Dumb Questions

Q: I have…questions. What…how do I put this…what exactly *is* Captain Amazing?

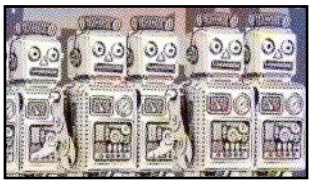
A: Captain Amazing is the world's most amazing object, the superhero protector of the innocent citizens of Objectville, and friend to all small animals everywhere.

More specifically, Captain Amazing is an anthropomorphized object, inspired by one of the most important comic book events of the early 21st century that deals with the death of a superhero—specifically, a comic that came out in 2007, when we were working on the first draft of *Head First C#* and looking for a good way to talk about the life and death of an object. We noticed a striking similarity between the shape of the objects in our memory heap diagrams and the shield of a certain famous comic book Captain…and thus, Captain Amazing was born. (If you're not a comic book fan, don't worry—you don't need to know anything about the comic we're making a reference to in order to understand the material in this chapter.)



Q: Why do your "clones" look like robots? Shouldn't they be people?

A: Yes, in our comic, we made the clones look like this:



because we didn't want to show graphic depictions of people getting destroyed.

Also, *relax*. The point of the comic panels in this chapter is to help get important C# and .NET concepts into your brain. The story is just a tool to draw some helpful analogies.

Your last chance to DO something... your object's finalizer

Sometimes you need to be sure something happens ***before*** your object gets garbage-collected, like releasing unmanaged resources.

In general, you'll never write a finalizer for an object that only owns *managed resources*. Everything you've encountered so far in this book has been managed by the CLR. But occasionally, programmers need to access a Windows resource that isn't in a .NET namespace. For example, if you find code on the internet—or an AI gives you code—that has `[DllImport]` above a declaration, you might be using an *unmanaged resource*. Some of those non-.NET resources might leave your system unstable if they're not "cleaned up." That's what finalizers are for.

A special method in your object called the **finalizer** allows you to write code that will always execute when your object is destroyed. It gets executed last, no matter what.

Let's do some experimentation with finalizers. **Create a new console app** and add this class with a finalizer:

```
class EvilClone
{
    public static int CloneCount = 0;
    public int CloneID { get; } = ++CloneCount;

    public EvilClone() => Console.WriteLine("Clone #{0} is wreaking havoc", CloneID);

    ~EvilClone()
    {
        Console.WriteLine("Clone #{0} destroyed", CloneID);
    }
}
```

When you put the ++ operator in front of a variable, it gets incremented before the statement is run. Why do you think we did that?

This is the *finalizer* (sometimes called a "destructor"). It's declared as a method that starts with a tilde ~ and has no return value or parameters. An object's finalizer is run *just before the object is garbage-collected*.

The top-level statements instantiate EvilClone objects, ***dereference*** (or remove references to) them, and collect them:

```
Console.WriteLine("Press a to add a clone, c to clear the list, and g to collect");
var stopwatch = System.Diagnostics.Stopwatch.StartNew();
var clones = new List<EvilClone>();
while (true)
{
    switch (Console.ReadKey(true).KeyChar)
    {
        case 'a':
            clones.Add(new EvilClone());
            break;
        case 'c':
            Console.WriteLine("Clearing list at time {0}", stopwatch.
ElapsedMilliseconds);
            clones.Clear();
            break;
        case 'g':
            Console.WriteLine("Collecting at time {0}", stopwatch.ElapsedMilliseconds);
            GC.Collect();
            break;
    };
}
```

We'll use *Stopwatch* to get an idea of how fast garbage collection runs. The Stopwatch class lets you accurately measure elapsed time by starting a new stopwatch and getting the number of milliseconds elapsed since you started it.

Run your app and press **a** a few times to create some EvilClone objects and add them to the List. Then press **c** to clear the List and remove all references to those EvilClone objects. Press **c** a few times—there's a *small* chance the CLR will collect some of the objects that were dereferenced, but you *probably* won't see them collected until you press **g** to call GC.Collect.

When EXACTLY does a finalizer run?

The finalizer for your object runs **after** all references are gone, but **before** that object gets garbage-collected. Garbage collection only happens after ***all*** references to your object go away, but it doesn't always happen *right after* the last reference disappears.

Suppose you have an object with a reference to it. The CLR sends the garbage collector to work, and it checks out your object. Since there are references to your object, the garbage collector ignores it and moves along. Your object keeps living on in memory.

Then, something happens. That last object holding a reference to *your* object removes that reference. Now your object is sitting in memory, with no references. It can't be accessed. It's basically a **dead object**.

But here's the thing: ***garbage collection is something that the CLR controls***, not your objects. So if the garbage collector isn't sent out again for, say, a few seconds, or maybe even a few minutes, your object still lives on in memory. It's unusable, but it hasn't been garbage-collected. **The object's finalizer cannot (yet) run.**

Finally, the CLR sends the garbage collector out again. It checks your object, finds there are no references, and runs the finalizer…possibly several minutes after the last reference to the object was removed or changed. Now that it's been finalized, your object is dead, and the collector tosses it away.

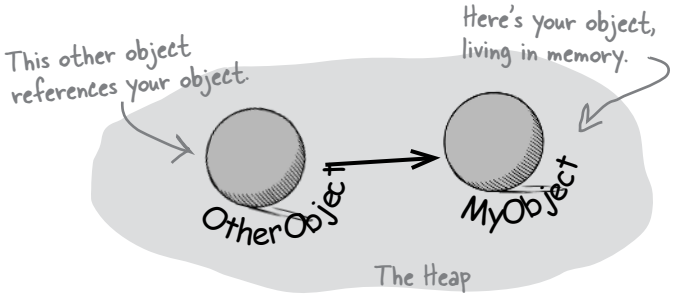
You can SUGGEST to .NET that it's time to collect the garbage

.NET does let you ***suggest*** that garbage collection would be a good idea. **Most times, you'll never use this method, because garbage collection is tuned to respond to a lot of conditions in the CLR, and calling it isn't really a good idea.** But just to see how a finalizer works, you could call for garbage collection on your own, using GC.Collect.

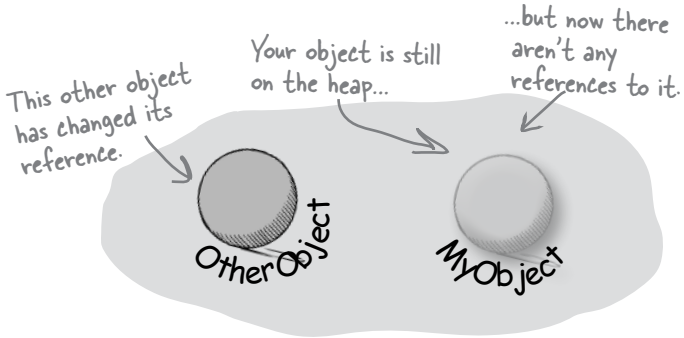
Be careful, though. That method doesn't ***force*** the CLR to garbage-collect things immediately. It just says, "Do garbage collection as soon as possible."

The life and death of an object...a timeline

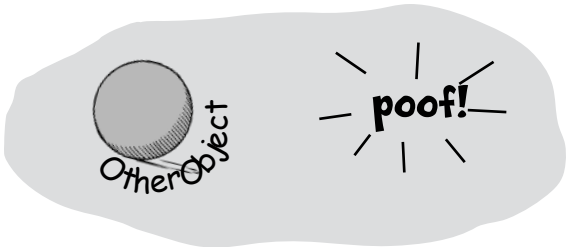
1. Your object is living its best life on the heap. Another object has a reference to it, keeping it alive.



2. The other object changes its reference, so now there are no other objects referencing your object.

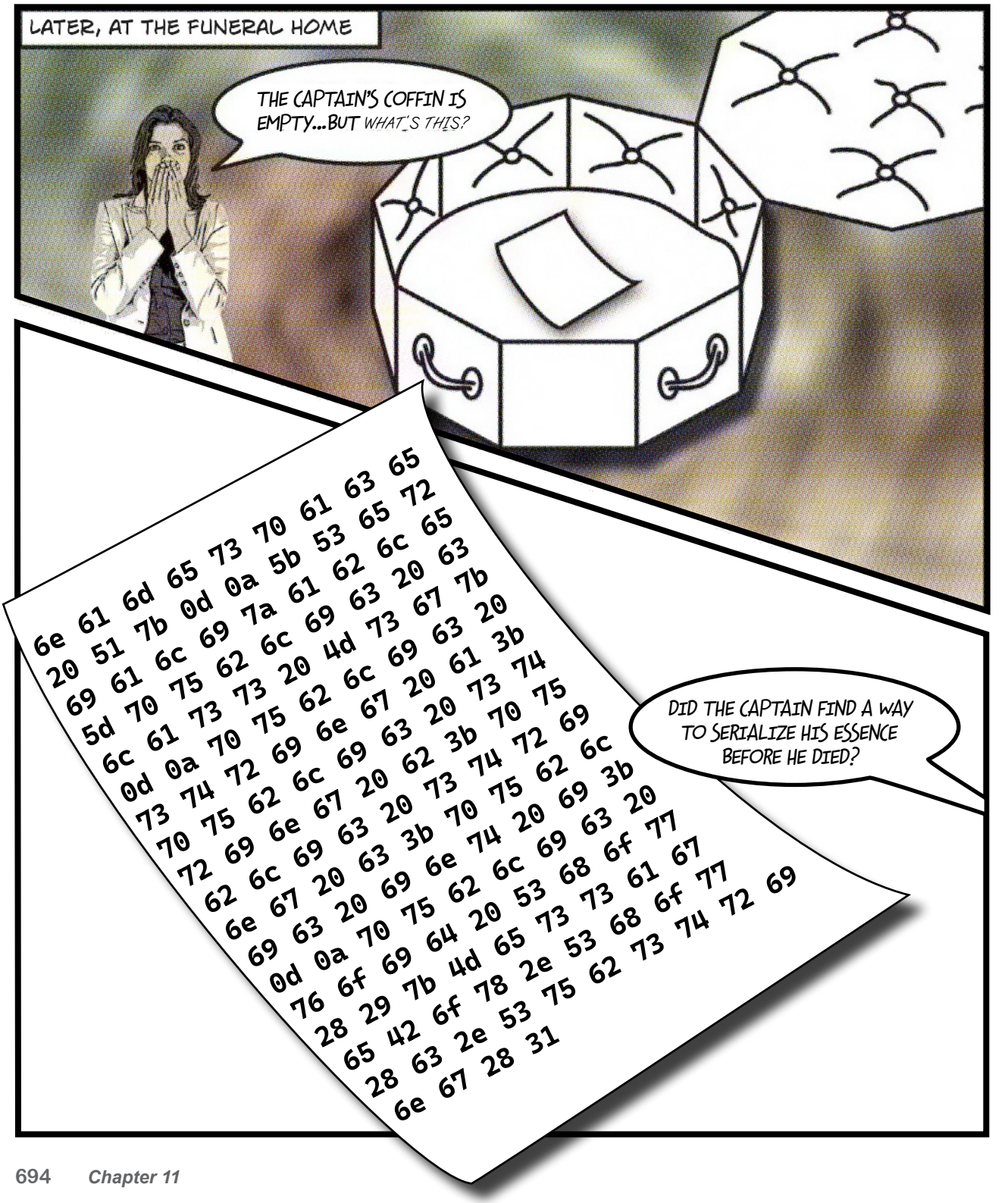


3. The CLR marks your object for garbage collection.



4. Eventually the garbage collector runs the object's finalizer and removes the object from the heap.

We're using GC.Collect as a learning tool to help you understand how garbage collection works. You definitely should not use it outside of toy programs (unless you really understand how garbage collection in .NET works on a deeper level than we'll go into in this book).


LATER, AT THE FUNERAL HOME
THE CAPTAIN'S COFFIN IS EMPTY...BUT WHAT'S THIS?
6e 61 6d 65 73 70 61 63 65
20 51 7b 0d 0a 5b 53 65 72
69 61 6c 69 7a 61 62 6c 65
5d 70 75 62 6c 69 63 20 63
6c 61 73 73 20 4d 73 67 7b
0d 0a 70 75 62 6c 69 63 20
73 74 72 69 6e 67 20 61 3b
70 75 62 6c 69 63 20 73 74
72 69 6e 67 20 62 3b 70 75
62 6c 69 63 20 73 74 72 69
6e 67 20 63 3b 70 75 62 6c
69 63 20 69 6e 74 20 69 3b
0d 0a 70 75 62 6c 69 63 20
76 6f 69 64 20 53 68 6f 77
28 29 7b 4d 65 73 73 61 67
65 42 6f 78 2e 53 68 6f 77
28 63 2e 53 75 62 73 74 72 69
6e 67 28 31
DID THE CAPTAIN FIND A WAY TO SERIALIZE HIS ESSENCE BEFORE HE DIED?

Finalizers can't depend on other objects

When you write a finalizer, you can't depend on it running at any one time. Even if you call GC.Collect, you're only suggesting that the garbage collector is run. It's not a guarantee that it'll happen right away. When it does, you have no way of knowing what order the objects will be collected in.

So what does that mean, in practical terms? Well, think about what happens if you've got two objects that have references to each other. If object #1 is collected first, then object #2's reference to it is pointing to an object that's no longer there. But if object #2 is collected first, then object #1's reference is invalid. That means you can't depend on references in your object's finalizer. That's why it's a really bad idea to try to do something inside a finalizer that depends on references being valid.

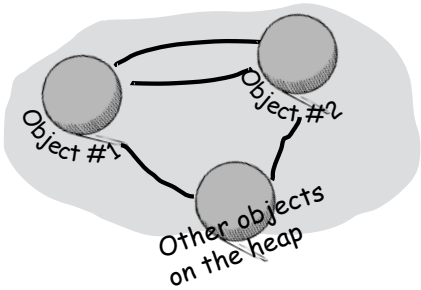
Don't use finalizers for serialization

Serialization is a really good example of something that you shouldn't do inside a finalizer. If your object's got a bunch of references to other objects, serialization depends on ***all*** of those objects still being in memory… and all of the objects they reference, and the ones those objects reference, and so on. So if you try to serialize when garbage collection is happening, you could end up missing vital parts of your program because some objects might've been collected ***before*** the finalizer ran.

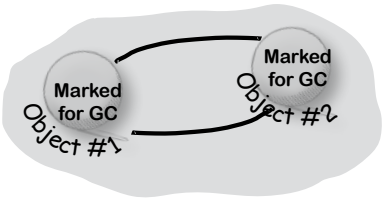
Luckily, C# gives us a really good solution to this: IDisposable. Anything that could modify your core data or that depends on other objects being in memory needs to happen as part of a Dispose method, not a finalizer.

Some people like to think of a finalizer as a kind of fail-safe for the Dispose method. That makes sense—you saw with your Clone object that just because you implement IDisposable, that doesn't mean the object's Dispose method will get called. However, you need to be careful. If your Dispose method depends on other objects that are on the heap, then calling Dispose from your finalizer can cause trouble. The best way around this is to make sure you always use a **`using`** statement any time you're creating an IDisposable object.

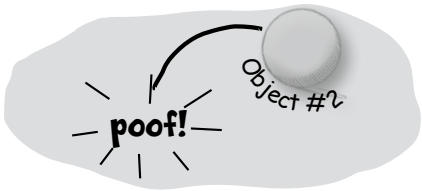
Start with two objects with references to each other.



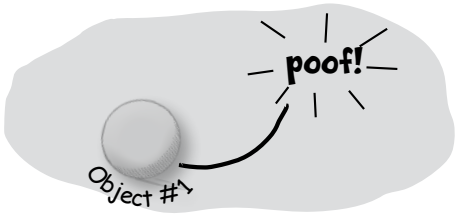
If all other objects on the heap remove their references to objects #1 and #2, they'll <u>both</u> be marked for collection.



If object #1 is collected first, then its data won't be available when the CLR runs object #2's finalizer.



On the other hand, object #2 could disappear before object #1. You've got no way of knowing the order.



And that's why one object's finalizer can't rely on any other object still being on the heap.

Fireside Chats

Tonight's talk: **the Dispose method and a finalizer spar over who's more valuable to you, a C# developer.**

Dispose:

To be honest, I'm a little surprised I was invited here. I thought the programming world had come to a consensus. I mean, I'm simply far more valuable as a C# tool than you are. Really, you're pretty feeble. You can't even depend on other objects still being around by the time that you're called. Pretty unstable, aren't you?

There's an interface specifically **because** I'm so important. In fact, I'm the only method in it!

OK, you're right, programmers need to know they're going to need me and either call me directly or use a `using` statement to call me. But they always know when I'll run, and they can use me to do whatever they need to do to clean up after their objects. I'm powerful, reliable, and easy to use. I'm a triple threat. And you? Nobody knows exactly when you'll run or what the state of the app will be when you finally do decide to show up.

You think you're a big shot because you always run with GC, but at least I can depend on other objects.

Finalizer:

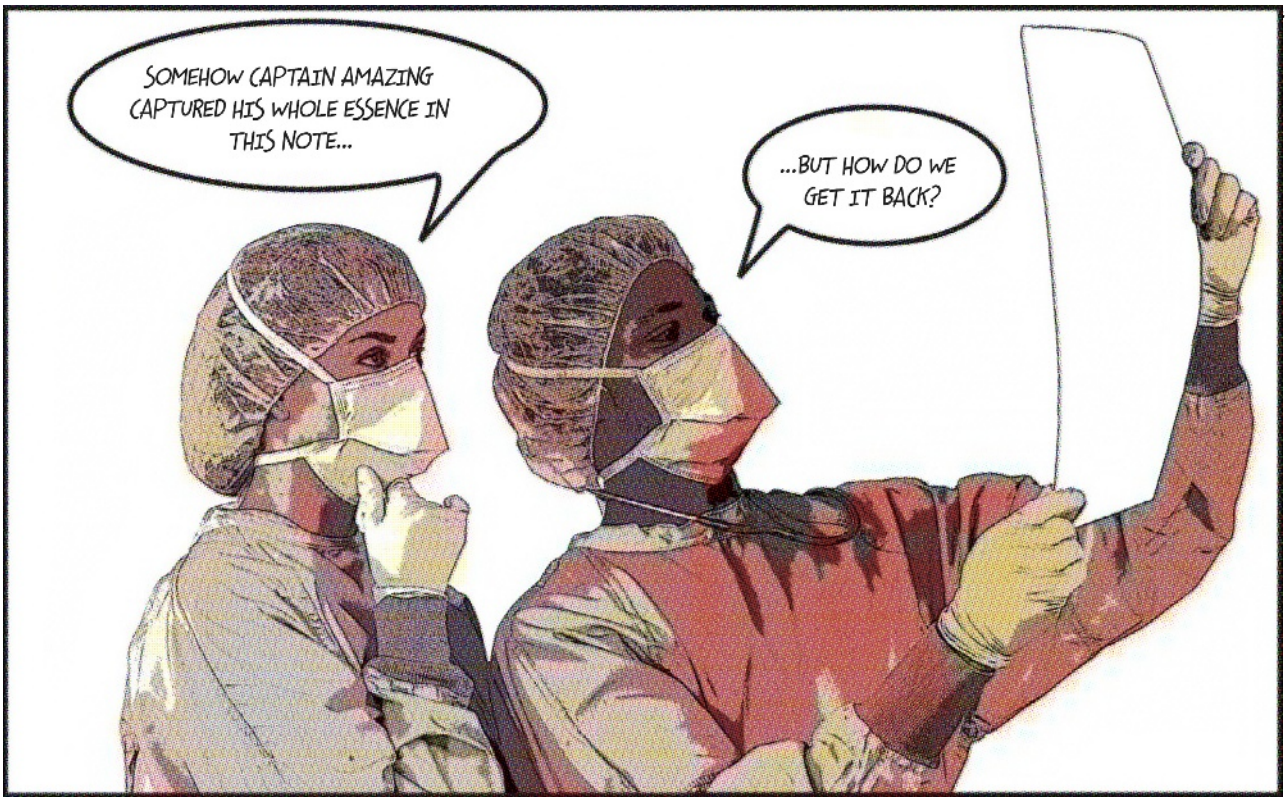
Excuse me? That's rich. I'm "feeble"? OK. Well, I didn't want to get into this, but since we're already stooping this low…at least I don't need an interface to get started. Without the IDisposable interface, well, let's face it... you're just another useless method.

Right, right…keep telling yourself that. And what happens when someone forgets to use a `using` statement when they instantiate their object? Then you're nowhere to be found.

Handles are what your programs use when they go around .NET and the CLR and interact directly with Windows. Since .NET doesn't know about them, it can't clean them up for you.

But if you need to do something at the very last moment just before an object is garbage-collected, there's no way to do it without me. I can free up network resources and Windows handles and anything else that might cause a problem for the rest of the app if you don't clean it up. I can make sure that your objects deal with being trashed more gracefully, and that's nothing to sneeze at.

That's right, pal—but I always run. You need someone else to run you. I don't need anyone or anything!



there are no Dumb Questions

Q: Can a finalizer use all of an object's fields and methods?

A: Yes. While you can't pass parameters to a finalizer method, you can use any of the fields in an object, either directly or using `this`—but be careful, because if those fields reference other objects, then the other objects may have already been garbage-collected. So your finalizer can call other methods and properties in the object…as long as those methods and properties *don't depend on other objects*.

Q: What happens to exceptions that get thrown in a finalizer?

A: Good question. Exceptions in finalizers work exactly like they do anywhere else in your code. Try replacing your EvilClone finalizer with this one that throws an exception:

```
~EvilClone() => throw new Exception();
```

Then run your app again, create some EvilClone instances, clear the list, and run the garbage collector. Your app will halt in the finalizer, just like it would if it hit any other exceptions. (Spoiler alert: in the next chapter, you'll learn about how to *catch* exceptions, a way to detect when they happen and run code to handle them.)

Q: How often does the garbage collector run automatically?

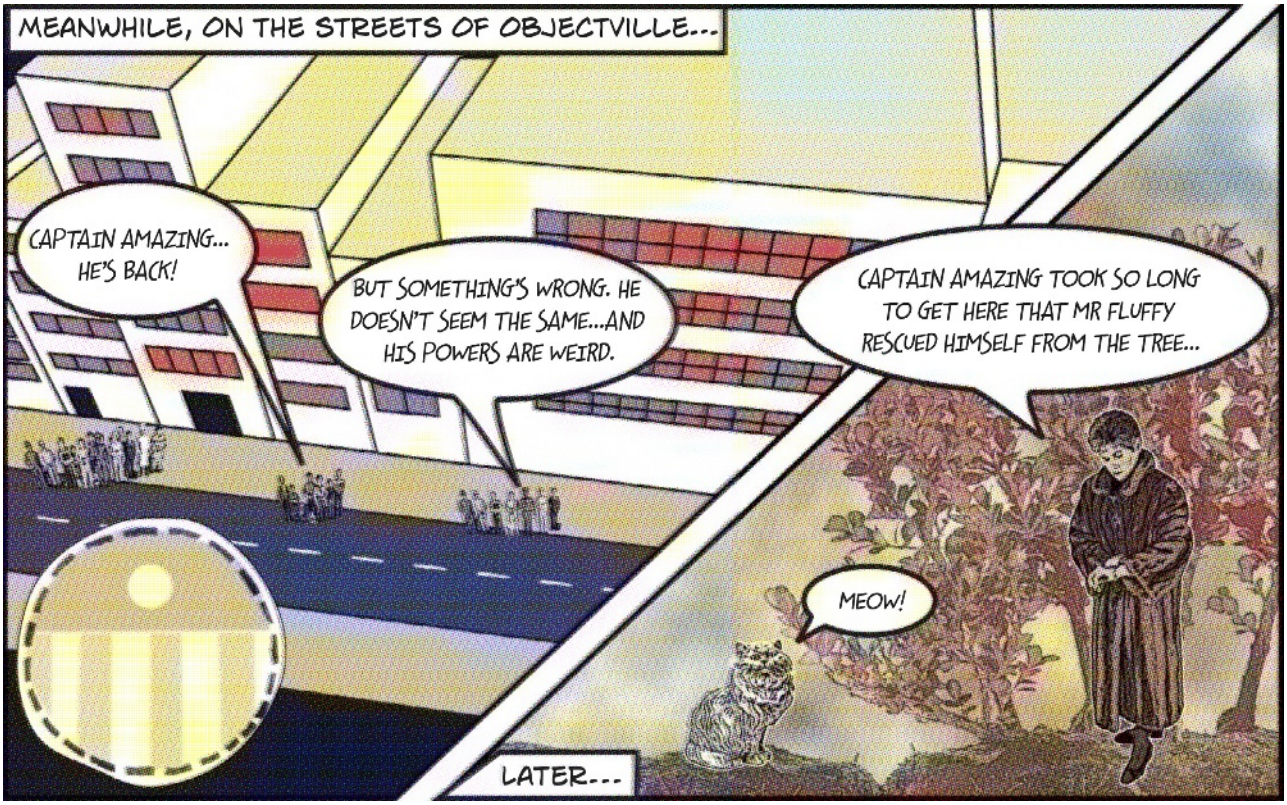
A: The short answer is: we're not sure. Garbage collection doesn't run on an easily predictable cycle, and you don't have firm control over it. You can be sure it will run when your program exits normally. Even when you call GC.Collect (which you generally should avoid), you're only suggesting that the CLR should start garbage collection.

Q: So how soon after I call GC.Collect will collection start?

A: When you run GC.Collect, you're telling .NET to garbage-collect as soon as possible. That's usually as soon as .NET finishes whatever it's doing. That means it'll happen pretty soon, but you can't precisely control when it starts.

Q: If something absolutely must run, do I put it in a finalizer?

A: No. It's possible that your finalizer won't run. It's possible to suppress finalizers when garbage collection happens. Or the process could end entirely. If you aren't freeing unmanaged resources, you're almost always better off using IDisposable and `using` statements.


MEANWHILE, ON THE STREETS OF OBJECTVILLE...
CAPTAIN AMAZING... HE'S BACK!
BUT SOMETHING'S WRONG. HE DOESN'T SEEM THE SAME...AND HIS POWERS ARE WEIRD.
CAPTAIN AMAZING TOOK SO LONG TO GET HERE THAT MR FLUFFY RESCUED HIMSELF FROM THE TREE...
MEOW!
LATER...

EVEN LATER...
Captain Amazing's Hideout Collection
TOP SECRET
PUFF...PANT...UGH! I'M EXHAUSTED.
WHAT'S WRONG? WHY ARE THE CAPTAIN'S POWERS BEHAVING DIFFERENTLY? IS THIS THE END?

A struct looks like an object...

We've been talking about the heap, because that's where your objects live. That's not the only part of memory where objects live. One of the types in .NET we haven't talked about much is the *struct*, and we'll use it to explore a different aspect of life and death in C#. Struct is short for **structure**, and structs look a lot like objects. They have fields and properties, just like objects. You can even pass them into a method that takes an object type parameter:

Structs can implement interfaces but can't subclass other classes. And structs are sealed, so they can't be extended.

A struct can have properties and fields...

...and define methods.

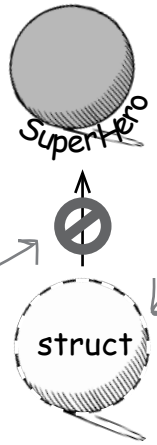
```
public struct AlmostSuperhero : IDisposable {
    private bool superStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, {0}, surrender now!",
                          villain.Name);
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.StartEpicBattle();
    }

    public void Dispose() => Console.WriteLine("Noooooooo!");
}
```

Structs can even implement interfaces like IDisposable.

...but isn't an object

But structs ***aren't*** objects. They *can* have methods and fields, but they *can't* have finalizers. They also can't inherit from classes or other structs, or have classes or structs inherit from them—you're allowed to use the : colon operator in a struct's declaration, but only if it's followed by one or more interfaces.

All structs extend System.ValueType, which in turn extends System.Object. That's why every struct has a ToString method—it gets it from Object. That's *the only inheriting* that structs are allowed to do.



You can mimic a standalone object with a struct, but structs don't stand in very well for complex inheritance hierarchies.

Structs can't inherit from other objects.

The power of objects lies in their ability to mimic real-world behavior, through inheritance and polymorphism.

Structs are best used for storing data, but the lack of inheritance and references can be a serious limitation.

Values get copied; references get assigned

We've seen how important references are to garbage collection—reassign the last reference to an object, and it gets marked for collection. We also know that those rules don't quite make sense for values. If we want to get a better sense of how objects and values live and die in the CLR's memory, we'll need to take a closer look at values and references: how they're similar, and more importantly, how they're different.

You already have a sense of how some types are different from others. On one hand you've got **value types** like int, bool, and decimal. On the other hand, you've got **objects** like List, Stream, and Exception. Types and objects don't quite work exactly the same way, do they?

When you use the equals sign to set one value type variable to another, it **makes a copy of the value**, and afterward, the two variables aren't connected to each other. On the other hand, when you use the equals sign with references, what you're doing is **pointing both references at the same object**.

- Variable declaration and assignment work the same with value types and object types:

```
int howMany = 25;
bool Scary = true;
List<double> temps = new List<double>();
throw new NotImplementedException();
```

int and bool are value types, List and Exception are object types.

- But once you start assigning values, you can see how they're different. Value types all are handled with copying. Here's an example—this should be familiar stuff:

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine("howMany has {0}, fifteenMore has {1}",
                  howMany, fifteenMore);
```

This line copies the value that's stored in the fifteenMore variable into the howMany variable and adds 15 to it.

Changing the fifteenMore variable has no effect on howMany, and vice versa.

The output here shows that `fifteenMore` and `howMany` are ***not*** connected:

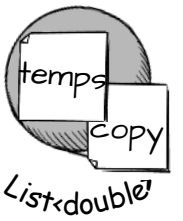
```
howMany has 25, fifteenMore has 40
```

- But as we know, when it comes to objects you're assigning references, not values:

```
temps.Add(56.5D);
temps.Add(27.4D);
List<double> copy = temps;
copy.Add(62.9D);
```

This line sets the copy reference to point to the same object as the temps reference.

Both references point at the same actual object.



So changing the List means both references see the update, since they both point to a single List object. Check this by writing a line of output:

```
Console.WriteLine("temps has {0}, copy has {1}", temps.Count(), copy.Count());
```

The output here demonstrates that `copy` and `temps` are actually pointing to the ***same*** object:

```
temps has 3, copy has 3
```

When you called copy.Add, it added a new temperature to the object that both copy and temps point to.

Structs are value types; objects are reference types

Let's take a closer look at how structs work, so you can start to understand when you might want to use a struct versus an object. When you create a struct, you're creating a **value type**. What that means is when you use equals to set one struct variable equal to another, you're creating a fresh *copy* of the struct in the new variable. So even though a struct *looks* like an object, it doesn't act like one.

Do this!

1 **Create a struct called Dog.**

Here's a simple struct to keep track of a dog. It looks just like an object, but it's not. Add it to a **new console application**:

```
struct Dog(string name, string breed)
{
    public string Name { get; set; } = name;

    public string Breed { get; set; } = breed;

    public void Speak() {
        Console.WriteLine("My name is {0} and I'm a {1}.", Name, Breed);
    }
}
```

Structs can have primary constructors, just like classes.

Use the primary constructor arguments to set the initial values for these automatic properties.

2 **Create a class called Canine.**

Make an exact copy of the Dog struct, but turn it into a class by **replacing the declaration**:

```
class Canine(string name, string breed)
```

Just change the first line to change it into a class named Canine, but keep the rest of the code in the class the same. Now you'll have a Canine *class* that has exactly the same code as the Dog *struct*.

3 **Add top-level statements that make copies of Dog and Canine data.**

Here's the code for the top-level statements:

```
Canine spot = new Canine("Spot", "pug");
Canine bob = spot;
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
betty.Name = "Betty";
betty.Breed = "pitbull";
jake.Speak();
```

MINI Sharpen your pencil

4 **Before you run the program…**

Write down what you think will be written to the console when you run this code:

.....

.....

MINI Sharpen your pencil Solution

What did you think would get written to the console?

My name is Spike and I'm a beagle.

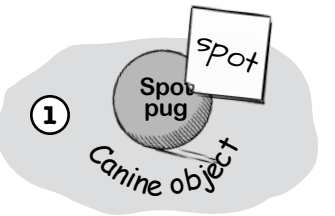
My name is Jake and I'm a poodle.

Here's what happened...

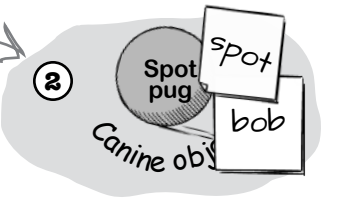
The **bob** and **spot** references both point to the same object, so both changed the same fields and accessed the same Speak method. Structs don't work that way. When you created **betty**, you made a **fresh copy of the data** in **jake**. The two structs are completely independent of each other.

```
Canine spot = new Canine("Spot", "pug");   (1)
Canine bob = spot;                         (2)
bob.Name = "Spike";                        (3)
bob.Breed = "beagle";
spot.Speak();
```

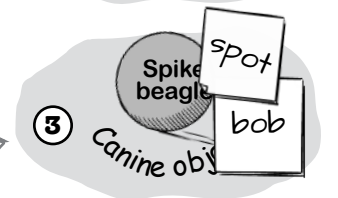
A new Canine object was created and the spot reference points to it.



The new reference variable bob was created, but no new object was added to the heap—the bob variable points to the same object as spot.

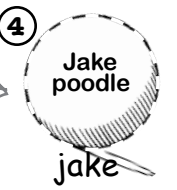


Since spot and bob both point to the same object, spot.Speak and bob.Speak both call the same method, and both of them produce the same output with "Spike" and "beagle."



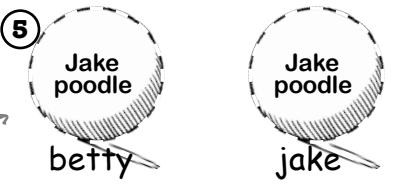
```
Dog jake = new Dog("Jake", "poodle");   (4)
Dog betty = jake;                       (5)
betty.Name = "Betty";                   (6)
betty.Breed = "pitbull";
jake.Speak();
```

When you create a new struct, it looks really similar to creating an object—you've got a variable that you can use to access its fields and methods.

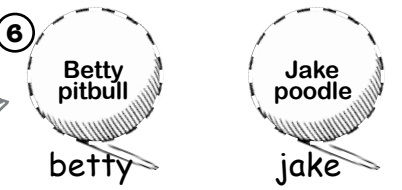


When you set one struct equal to another, you're creating a fresh COPY of the data inside the struct. That's because struct is a VALUE TYPE (not an object or reference type).

Here's the big difference. When you added the betty variable, you created a whole new value.



Since you created a fresh copy of the data, jake was unaffected when you changed betty's fields.



The stack vs. the heap: more on memory

Behind the Scenes

Let's quickly recap how a struct differs from an object. You've seen that you can make a fresh copy of a struct just using equals, which you can't do with an object. But what's really going on behind the scenes?

The CLR divides your data between two places in memory: the heap and the stack. You already know that objects live on the **heap**. The CLR also reserves another part of memory called the **stack**, where it stores the local variables you declare in your methods and the parameters that you pass into those methods. You can think of the stack as a bunch of slots that you can stick values in. When a method gets called, the CLR adds more slots to the top of the stack. When it returns, its slots are removed.

Remember, when your program's running, the CLR is actively managing memory, dealing with the heap, and collecting garbage.

The Code

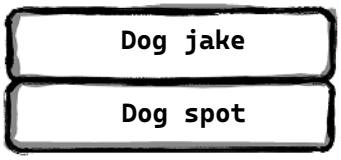
Here's some code you might see in a program.

The Stack

This is where structs and local variables hang out.

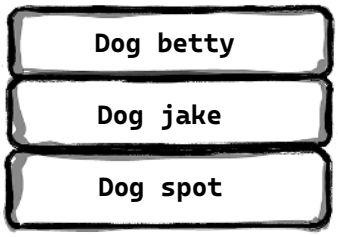
```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
```

Here's what the stack looks like after these two lines of code run.



```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
```

When you create a new struct—or any other value-type variable—a new "slot" gets added onto the stack. That slot is a copy of the value in your type.



```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
SpeakThreeTimes(jake);
```

```
public SpeakThreeTimes(Dog dog) {
    int i;
    for (i = 0; i < 5; i++)
        dog.Speak();
}
```

When you call a method, the CLR puts its local variables on the top of the stack. In this case, the code calls the SpeakThreeTimes method. It has one parameter (dog) and one variable (i), and the CLR stores them on the stack. When the method returns, the CLR pops i and dog off the stack—and that's how values live and die in the CLR.

This all sounds theoretical, but I bet there's a good practical reason to learn about the stack, the heap, values, and references.

It's important to understand how a struct you copy by value is different from an object you copy by reference.

There are times when you need to be able to write a method that can take either a value type ***or*** a reference type—perhaps a method that can work with either a Dog struct or a Canine object. If you find yourself in that situation, you can use the **`object`** keyword:

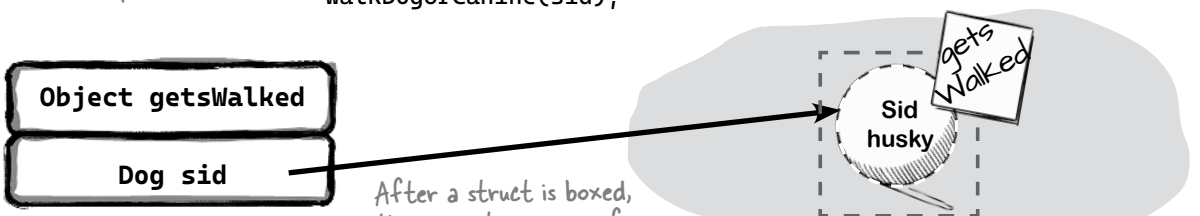
```
   public void WalkDogOrCanine(object getsWalked) { ...
}
```

If you send this method a struct, the struct gets **boxed** into a special object "wrapper" that allows it to live on the heap. While the wrapper's on the heap, you can't do much with the struct. You have to "unwrap" the struct to work with it. Luckily, all of this happens *automatically* when you set an object equal to a value type, or pass a value type into a method that expects an object.

You can also use the "is" keyword to see if an object is a struct, or any other value type, that's been boxed and put on the heap.

1 Here's what the stack and heap look like after you create an object variable and set it equal to a Dog struct.

```
Dog sid = new Dog("Sid", "husky");
WalkDogOrCanine(sid);
```

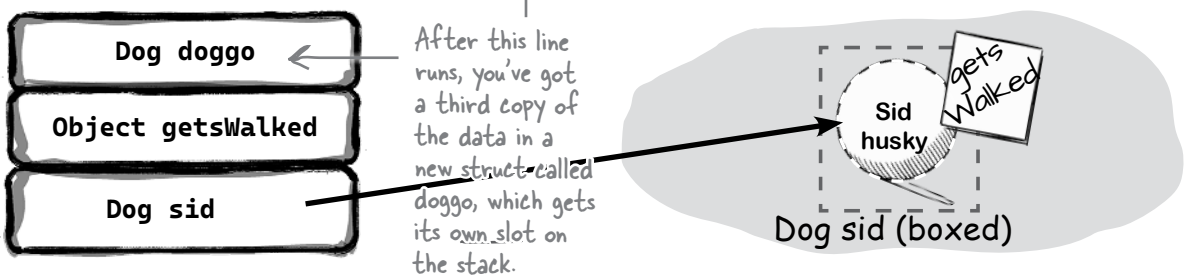


After a struct is boxed, there are two copies of the data: the copy on the stack, and the copy boxed on the heap.

The WalkDogOrCanine method takes an object reference, so the Dog struct was boxed before it was passed in. Casting it back to a Dog unboxes it.

2 If you want to **unbox the object**, all you need to do is cast it to the right type, and it gets unboxed automatically. The **`is`** keyword works just fine with structs, but be careful, because the **`as`** keyword doesn't work with value types.

```
if (getsWalked is Dog doggo) doggo.Speak();
```



After this line runs, you've got a third copy of the data in a new struct called doggo, which gets its own slot on the stack.

Behind the Scenes

When a method is called, it looks for its arguments on the stack.

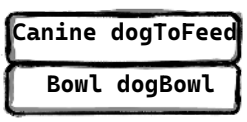
The stack plays an important part in how the CLR manages your app's data. One thing we take for granted is the fact that we can write a method that calls another method, which in turn calls another method. In fact, a method can call itself (which is called *recursion*). The stack is what gives our programs the ability to do that.

Here are three methods from a dog simulator program. The FeedDog method calls the Eat method, which calls the CheckBowl method.

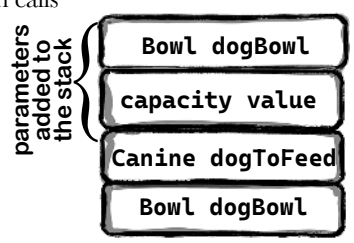
```
public double FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return dogToFeed.MealSize + .05D; // A little is always
spilled
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5D) {
        string message = "My bowl's almost empty!";
        Console.WriteLine(message);
    }
}
```

Remember the terminology here: a "parameter" specifies the values a method needs; an "argument" is the actual value or reference that you pass into a method when you call it.

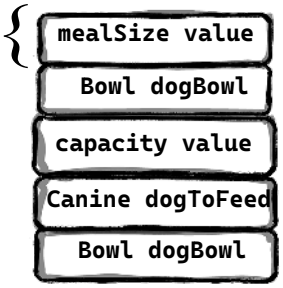
Here's what the stack looks like as FeedDog calls Eat, which calls CheckBowl, which in turn calls Console.WriteLine().



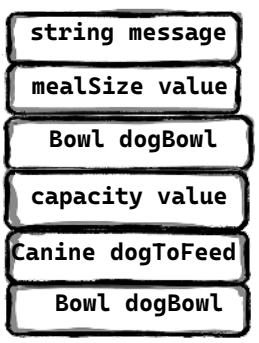
The FeedDog method takes two parameters, a Canine reference and a Bowl reference. So when it's called, the two arguments passed to it are on the stack.



FeedDog needs to pass two arguments to the Eat method, so they're pushed onto the stack as well.



As the method calls pile up and the program goes deeper into methods that call methods that call other methods, the stack gets bigger and bigger.



When WriteLine exits, its arguments will be popped off of the stack. That way, Eat can keep going as if nothing had happened.

```
Vector3 vector3 = Vector3.zero;
```

struct UnityEngine.Vector3
Representation of 3D vectors and points.

GitHub Examples and Documentation (Ctrl+Alt+')

Since Unity's Vector3 is a struct, creating a bunch of vectors won't cause extra GCs.

Go to a Unity project and hover over Vector3—it's a struct. Garbage collection (or GC) can seriously slow down an app's performance, and a lot of object instances in your game could trigger extra GCs and slow down the frame rate. Games often use a LOT of vectors. Making them structs means their data is kept on the stack, so even creating millions of vectors won't cause extra GCs that will slow down your game.

Use out parameters to make a method return more than one value

Do this!

Speaking of parameters and arguments, there are a few more ways that you can get values into and out of your programs. They all involve adding **modifiers** to your method declarations. One of the most common ways of doing this is by using the **`out` modifier** to specify an output parameter. You've seen the `out` modifier many times—you use it every time you call the int.TryParse method. You can also use the `out` modifier in your own methods. Create a new console app and add this empty method declaration to the form. Note the `out` modifiers on both parameters:

```
public static int ReturnThreeValues(int value, out double half, out int twice)
{
    return value + 1;
}
```

The out parameter 'half' must be assigned to before control leaves the current method

The out parameter 'twice' must be assigned to before control leaves the current method

Show potential fixes (Alt+Enter or Ctrl+.)

Take a closer look at those two errors:

- *The out parameter 'half' must be assigned to before control leaves the current method*
- *The out parameter 'twice' must be assigned to before control leaves the current method*

Any time you use an `out` parameter, you ***always*** need to set it before the method returns—just like you always need to use a `return` statement if your method is declared with a return value.

Here's all of the code for the app:

```
int ReturnThreeValues(int value, out double half, out int twice)
{
    half = value / 2f;
    twice = value * 2;
    return value + 1;
}

Console.Write("Enter a number: ");
if (int.TryParse(Console.ReadLine(), out int input))
{
    var output1 = ReturnThreeValues(input, out double output2, out int output3);

    Console.WriteLine("Outputs: plus one = {0}, half = {1:F}, twice = {2}",
        output1, output2, output3);
}
```

All out parameters must be assigned before the method exits.

Here's the familiar code you've used throughout the book that uses the out modifier with int.TryParse to convert a string to an int.

You'll also use the out modifier when you call your new method.

Here's what it looks like when you run the app:

```
Enter a number: 17
Outputs: plus one = 18, half = 8.50, twice = 34
```

Pass by reference using the ref modifier

One thing you've seen over and over again is that every time you pass an int, double, struct, or any other value type into a method, you're passing a copy of that value to that method. There's a name for that: **pass by value**, which means that the entire value of the argument is copied.

But there's another way to pass arguments into methods, and it's called **pass by reference**. You can use the **`ref`** keyword to allow a method to work directly with the argument that's passed to it. Just like the **`out`** modifier, you need to use **`ref`** when you declare the method and also when you call it. It doesn't matter if it's a value type or a reference type either—any variable that you pass to a method's **`ref`** parameter will be directly altered by that method.

An "out" argument is just like a "ref" argument, except that it doesn't need to be assigned before going into the method, and must be assigned before the method returns.

To see how it works, create a new console app with this Guy class and these methods:

```
void ModifyAnIntAndGuy(ref int valueRef, ref Guy guyRef)
{
    valueRef += 10;
    guyRef.Name = "Bob";
    guyRef.Age = 37;
}
```

When this method sets valueRef and guyRef, what it's really doing is changing the values of the variables in the method that called it.

This class uses an object initializer instead of a primary constructor. You'll see why shortly.

```
var i = 1;
var guy = new Guy() { Name = "Joe", Age = 26 };
Console.WriteLine("i is {0} and guy is {1}", i, guy);
ModifyAnIntAndGuy(ref i, ref guy);
Console.WriteLine("Now i is {0} and guy is {1}", i, guy);
```

When the top-level statements call ModifyAnIntAndGuy, it passes its i and guy variables *by reference*. The method uses them like any other variable. But because they were passed by reference, the method was actually updating the original variables all along, and *not just copies of them on the stack*. When the method exits, the i and guy variables in the top-level statements are updated directly.

Don't add a question mark ? after "string" like you usually do. The compiler will give you a warning, but the code will still run.

```
class Guy
{
    public string Name { get; set; }
    public int Age { get; set; }
    public override string ToString() =>
                        $"a {Age}-year-old named {Name}";
}
```

Run your app—it writes this output to the console:

```
i is 1 and guy is a 26-year-old named Joe
Now i is 11 and guy is a 37-year-old named Bob
```

The second line is different from the first because the ModifyAnIntAndGuy modified references to the variables in the top-level statements.

Value types have a TryParse method that uses out parameters

You've been using int.TryParse to convert strings to int values ("parsing" means analyzing text and extracting values). Other value types have similar functions: double.TryParse will attempt to convert strings to double values, bool.TryParse will do the same for Boolean values, and so on for decimal.TryParse, float.TryParse, long.TryParse, byte.TryParse, and more. Remember back in Chapter 10 when we used a `switch` statement to convert the string "Spades" into a Suits.Spades enum value? Well, the static Enum.TryParse method does the same thing, except for enums.

Use optional parameters to set default values

A lot of times, your methods will be called with the same arguments over and over again, but the method still needs the parameter because occasionally it changes. It would be useful if you could set a *default value*, so you only needed to specify the argument when calling the method if it was different.

That's exactly what **optional parameters** do. You can specify an optional parameter in a method declaration by using an equals sign followed by the default value for that parameter. You can have as many optional parameters as you want, but all of the optional parameters have to come after the required parameters.

Here's an example of a method that uses optional parameters to check if someone has a fever:

```
void CheckTemperature(double temp, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temp < tooHigh && temp > tooLow)
        Console.WriteLine("{0} degrees F - feeling good!", temp);
    else
        Console.WriteLine("Uh-oh {0} degrees F -- better see a doctor!", temp);
}
```

Optional parameters have default values specified in the declaration.

This method has two optional parameters: **`tooHigh`** has a default value of 99.5, and **`tooLow`** has a default value of 96.5. Calling CheckTemperature with one argument uses the default values for both **`tooHigh`** and **`tooLow`**. If you call it with two arguments, it will use the second argument for the value of **`tooHigh`**, but still use the default value for **`tooLow`**. You can specify all three arguments to pass values for all three parameters.

If you want to use some (but not all) of the default values, you can use **named arguments** to pass values for just those parameters that you want to pass. All you need to do is give the name of each parameter followed by a colon and its values. If you use more than one named argument, make sure you separate them with commas, just like any other arguments.

Add the CheckTemperature method to a console app, then add these top-level statements:

```
// Those values are fine for your average person
CheckTemperature(101.3);

// A dog's temperature should be between 100.5 and 102.5 Fahrenheit
CheckTemperature(101.3, 102.5, 100.5);

// Bob's temperature is always a little low, so set tooLow to 95.5
CheckTemperature(96.2, tooLow: 95.5);
```

It prints this output, working differently based on different values for the optional parameters:

```
Uh-oh 101.3 degrees F -- better see a doctor!
101.3 degrees F - feeling good!
96.2 degrees F - feeling good!
```

Use optional parameters and named arguments when you want your methods to have default values.

A null reference doesn't refer to any object

Do this!

When you create a new reference and don't set it to anything, it has a value. It starts off set to **`null`**, which means it's not pointing to anything. Let's experiment with null references.

1. **Create a new console app** and add the Guy class you used to experiment with the **`ref`** keyword.

2. Then **add this code** that creates a new Guy object but *doesn't set its Name property*:

```
Guy guy;
guy = new Guy() { Age = 25 };
Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
```

Leaving Name out of the object initializer initializes it to its default value, which for a string is null.

The compiler is giving you a warning that the Name property may be null. That's a pretty good warning to look out for!

string? Guy.Name { get; set; }

'Name' may be null here.

CS8602: Dereference of a possibly null reference.

3. **Place a breakpoint** on the last line of the top-level statements and debug your app. When it hits the breakpoint, **hover over `guy.Name`** to inspect its value:

```
1  Guy guy;
2  guy = new Guy() { Age = 25 };
3  Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
```

guy.Name null

String is a reference type. Since you didn't set its value in the Guy object, it still has its default value: null.

4. **Continue running the code.** Console.WriteLine tries to access the Length property of the String object referenced by the guy.Name property, and throws an exception:

```
3  Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
4
5
6
7
```

Exception Thrown

System.NullReferenceException: 'Object reference not set to an instance of an object.'

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings

☑ Break when this exception type is thrown

Except when thrown from:

☐ ConsoleApp30.dll

Open Exception Settings | Edit Conditions

When the CLR throws a *NullReferenceException* (which developers often refer to as an *NRE*), it's telling you that it tried to access a member of an object, but the reference that it used to access that member was null. Developers *try to prevent* null reference exceptions.

Can you think of how you would prevent that null reference exception?

Non-nullable reference types help you avoid NREs

The easiest way to avoid null reference exceptions (or NREs) is to **design your code so references can't be null**. Luckily, the C# compiler gives you a really useful tool to help deal with nulls by warning you when a value may be null. Take another look at that compiler warning you got when you used `string` instead of `string?` in your Guy class:

> ⚠ CS8618 Non-nullable property 'Name' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.

The C# compiler did something really interesting: it used *flow analysis* (or a way of analyzing the various paths through the code) to determine that ***it's possible for the Name property to be assigned a null value***.

That warning is telling you that your code could potentially throw a NullReferenceException when any code is run that **dereferences** the Name property, which is what it's called when the CLR tries to look up the object that a reference is pointing to. String is a reference type, so its value is held in an object somewhere on the heap. The Name property just contains a reference to that object. When the CLR tries to figure out what string that reference is pointing to, it finds a null instead, and throws the NullReferenceException.

So what should we do about it?

Use encapsulation to prevent your property from ever being null

Take another look at that warning message—it's actually a suggestion for how to fix it: make sure the Name property has a non-null value when the constructor ends. You *could* assign it a default value:

```
public string Name { get; private set; } = "";
```

But that's not a great solution if guys should always have a name.

Back in Chapter 5, you learned all about how to use encapsulation to keep your class members from having invalid values. So go ahead and make the Name property private, then add a primary constructor, and add an expression-bodied member for the Name property (just like you did with Guy, Outfit, and HairColor when you were experimenting with JSON serialization in Chapter 10):

```
public string Name => name;
```

What if you want to be able to set the Age? No problem. Use a property with get and set accessors. You can even use lambda expressions for them. Then use the age argument to set the value:

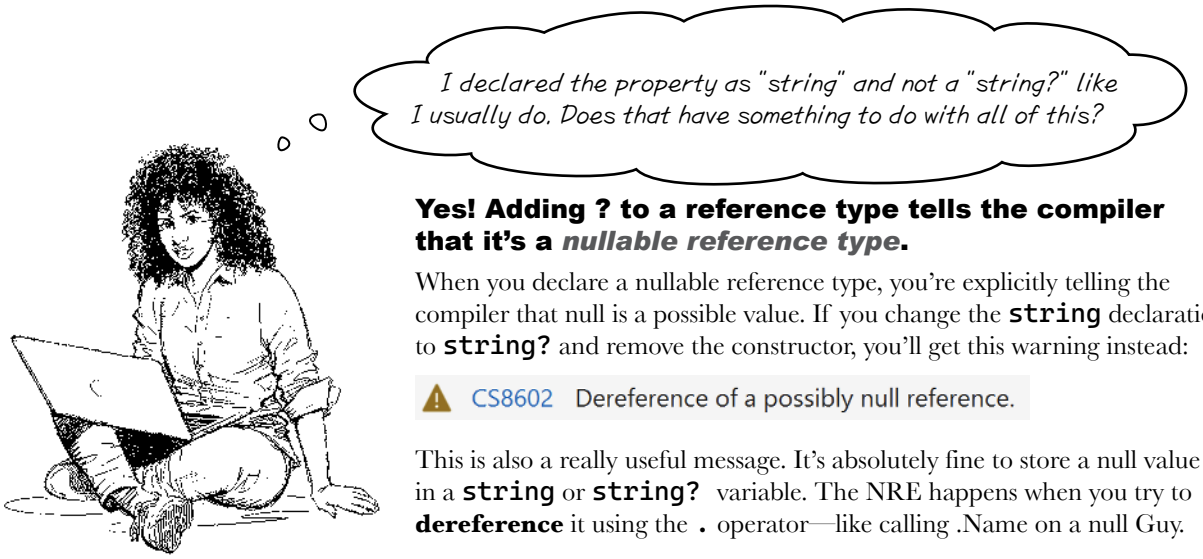
```
public int Age { get => age; set => age = value; }
```

Individual get and set accessors can have expression bodies.

Here's the final Guy class, but now it uses encapsulation to make sure `Name` is never null:

```
class Guy(string name, int age)
{
    public string Name => name;
    public int Age { get => age; set => age = value; }
    public override string ToString() => $"a {age}-year-old named {name}";
}
```

Once you encapsulate the Name property with a constructor that always sets it to a non-null value, you prevent it from ever being set to null, and the warning disappears. However, if you try to call that constructor with a potentially null value, you'll get the same compiler warning on that code instead.


I declared the property as "string" and not a "string?" like I usually do. Does that have something to do with all of this?

Yes! Adding ? to a reference type tells the compiler that it's a *nullable reference type*.

When you declare a nullable reference type, you're explicitly telling the compiler that null is a possible value. If you change the `string` declaration to `string?` and remove the constructor, you'll get this warning instead:

⚠ CS8602 Dereference of a possibly null reference.

This is also a really useful message. It's absolutely fine to store a null value in a `string` or `string?` variable. The NRE happens when you try to **dereference** it using the `.` operator—like calling .Name on a null Guy.

Sharpen your pencil

You've been using nullable reference types throughout the book, but haven't gotten the warnings we just saw. Why doesn't this code from Chapter 3 give a "possibly null reference" warning?

```
Console.Write("Enter the number of cards to pick: ");
string? line = Console.ReadLine();
if (int.TryParse(line, out int numberOfCards))
{
    string[] cards = CardPicker.PickSomeCards(numberOfCards);
    foreach (string card in cards)
    {
        Console.WriteLine(card);
    }
}
```

In Chapter 8, you implemented the IComparer<Duck> interface, which included a Compare method with two nullable Duck? parameters, x and y. Why doesn't this code give a "possibly null reference" warning?

```
class DuckComparerByKind : IComparer<Duck> {
   public int Compare(Duck? x, Duck? y) {
       if (x == null || y == null) return 0;

       if (x.Kind < y.Kind)
           return -1;
       if (x.Kind > y.Kind)
           return 1;
       else
           return 0;
        }
}
```

Sharpen your pencil Solution

Why doesn't this code from Chapter 3 give a "possibly null reference" warning?

```
Console.Write("Enter the number of cards to pick: ");
string? line = Console.ReadLine();
if (int.TryParse(line, out int numberOfCards))
{
    string[] cards = CardPicker.PickSomeCards(numberOfCards);
    foreach (string card in cards)
    {
        Console.WriteLine(card);
    }
}
```

The int.TryParse method takes a string? parameter, so there's no place in the code that the . operator is used to dereference the line variable.

In Chapter 8, you implemented the IComparer<Duck> interface, which included a Compare method with two nullable Duck? parameters, x and y. Why doesn't this code give a "possibly null reference" warning?

```
class DuckComparerByKind : IComparer<Duck> {
   public int Compare(Duck? x, Duck? y) {
       if (x == null || y == null) return 0;

       if (x.Kind < y.Kind)
           return -1;
       if (x.Kind > y.Kind)
           return 1;
       else
           return 0;
        }
}
```

The x and y parameters are both declared as nullable Duck references. But the method checks them both to make sure that they aren't null before dereferencing them with the . operator, so that code will only run if both x and y contain valid non-null references.

Bullet Points

- You can use the **GC class** to suggest that the CLR runs garbage collection, but it's not a tool most developers need to use beyond learning and experimentation.
- Use ~ to define a **finalizer** method in your class to clean up unmanaged resources.
- A finalizer can't depend on the rest of the object graph being available, which is why it's a bad idea to use for serialization.
- A **struct** is a value type that has fields, properties, and methods.
- You can use = to copy a struct and == to compare two structs because they're value types.
- Use the **out modifier** to define out parameters to make a method return more than one value.
- Use the **ref modifier** to pass an argument by reference, so the method can modify the original value or reference that it's called with.
- Use **optional parameters** to set default values.
- When a reference is null, attempting to dereference it by using its methods, properties, or fields will cause the CLR to throw a **NullReferenceException**.
- When you don't use ? to declare a reference type, it becomes **non-nullable**, which causes the compiler to warn you if it can potentially be assigned a null value.

Nullable value types can be null...and handled safely

When you declare an int, bool, or another value type, if you don't specify a value, the CLR assigns it a default value like 0 or true. But let's say you're writing code to store data from a survey where there's a yes/no question that's optional. What if you need to represent a Boolean value that could be true or false, or not have a value at all?

That's where **nullable value types** can be very useful. A nullable value type can either have a value or be set to null. It takes advantage of a generic struct Nullable<T> that can be used to ***wrap*** a value (or contain the value and provide members to access and work with it). If you set a nullable value type to null, it doesn't have a value—and Nullable<T> gives you handy members to let you work safely with it *even in this case*.

You can declare a nullable Boolean value like this:

```
Nullable<bool> optionalYesNoAnswer = null;
```

C# also has a shortcut—for a value type T, you can declare Nullable<T> like this: **T?**

```
bool? anotherYesNoAnswer = false;
```

The Nullable<T> struct has a property called Value that gets or sets the value. A bool? will have a value of type bool, an int? will have one of type int, etc. They'll also have a property called HasValue that returns true if it's not null.

Nullable<bool>
Value: DateTime HasValue: bool ...
GetValueOrDefault(): DateTime ...

You can always convert a value type to a nullable type:

```
int? myNullableInt = 9321;
```

And you can get the value back using its handy Value property:

```
int = myNullableInt.Value;
```

Nullable<T> is a struct that lets you store a value type OR a null value. Here are some of the methods and properties of Nullable<bool>.

But the Value call eventually just casts the value with `(int)myNullableInt`—and it will throw an InvalidOperationException if the value is null. That's why Nullable<T> also has a HasValue property, which returns true if the value is not null, and false if it is. You can also use the convenient GetValueOrDefault method, which safely returns a default value if the Nullable has no value. You can optionally pass it a default value to use, or use the type's normal default value.

T? is an alias for Nullable<T> (when T is a value type)

When you add a question mark to any value type (like int? or decimal?), the compiler translates that to the Nullable<T> struct (Nullable<int> or Nullable<decimal>). You can see this for yourself: add a Nullable<bool> variable to a program, put a breakpoint on it, and add a watch for it in the debugger. You'll see bool? displayed in the Watch window in the IDE. This is an example of an **alias**, and it's not the first one you've encountered. Hover your cursor over any int. You'll see that it translates to a struct called System.Int32:

```
int value = 3;
```

int.Parse() and int.TryParse() are members of this struct.

readonly struct System.Int32
Represents a 32-bit signed integer.

Take a minute and do that for each of the types at the beginning of Chapter 4. Notice how all of them are aliases for structs—except for string, which is a class called System.String, not a value type.

The null-coalescing operator ?? checks for nulls automatically

Sometimes you can't avoid working with nulls. For example, you learned about reading data from strings using StringReader in Chapter 10. Create a new console app and add this code:

```
using (var stringReader = new StringReader(""))
{
    var nextLine = stringReader.ReadLine();
    Console.WriteLine("Line length is: {0}", nextLine.Length);
}
```

The compiler warns you that nextLine may be null, and gives the warning about dereferencing a possibly null reference. It's right—when you run the code, it throws a NullReferenceException.

(local variable) string? nextLine

'nextLine' may be null here.

CS8602: Dereference of a possibly null reference.

?? checks for null and returns an alternative

One way to prevent a null reference from being accessed (or dereferenced) is to use the **null-coalescing operator ??** to evaluate the potentially null expression—in this case, calling stringReader.ReadLine—and returning an alternative value if it's null. Modify the first line of the `using` block to add `?? String.Empty` to the end of the line:

```
var nextLine = stringReader.ReadLine() ?? String.Empty;
```

And as soon as you add this, the warning goes away. That's because the null coalescing operator tells the C# compiler to execute stringReader.ReadLine; and use the value it returns if it's non-null, but substitute the value you provided (in this case, an empty string) if it is.

??= assigns a value to a variable only if it's null

When you're working with null values, it's really common to write code that checks if a value is null and assigns it a non-null value to avoid an NRE. For example, if you wanted to modify your program to print the first line of code, you might write this:

```
if (nextLine == null)
    nextLine = "(the first line is null)";

// Code that works with nextLine and needs it to be non-null
```

You can rewrite that conditional statement using the **null assignment ??=** operator:

```
nextLine ??= "(the first line was empty)";
```

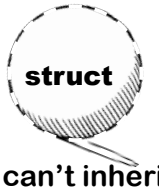
The ??= operator checks the variable, property, or field on the left side of the expression (in this case, nextLine) to see if it's null. If it is, the operator assigns the value on the right side of the expression to it. If not, it leaves the value intact.

String.Empty returns an empty string.

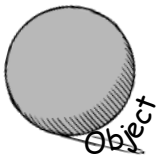
String.Empty is a static field on the String class that returns the empty string ("). It's declared in the String class, and used internally as a return value when its Length property equals zero.

"Captain" Amazing...not so much

You should have a pretty good idea by now of what was going on with the less-powerful, more-tired Captain Amazing. In fact, it wasn't Captain Amazing at all, but a boxed struct:



vs.



- **Structs can't inherit from classes.**
  No wonder the Captain's superpowers seemed a little weak! He didn't get any inherited behavior.

- **You can use the as keyword with an object.**
  Objects allow for polymorphism by allowing an object to function as any of the objects it inherits from.

- **Structs are copied by value.**
  This is one of the most useful things about them. It's especially useful for encapsulation.

  One important point: you can use the "is" keyword to check if a struct implements an interface, which is one aspect of polymorphism that structs do support.

- **You can't create a fresh copy of an object.**
  When you set one object variable equal to another, you're copying a ***reference*** to the ***same*** variable.

  That's one advantage of structs (and other value types)—you can easily make copies of them.

*Is String **really a class**, though? I use can use == to compare value types like int, float, and byte, if I use == with a reference type like Guy or Duck, it just tells me if they're pointing to the same object. So how can a string be an object?*

String really is a class. Any reference type can use == too.

The `string` keyword is an alias for the System.String type, a class that keeps track of strings made up of sequences of char values. The System.String class implements several familiar interfaces, including IEnumerable<char> (because under the hood it's a sequence of char values) and IComparable<string?>—these are the same interfaces you learned about in Chapters 8 and 9.

It also implements the **IEquatable<string> interface**, which has a single method, Equals(T) that compares the current object with another and returns true if they're equal or false if they're not—actually really similar to IComparable<T>, which compares objects to determine their order. Collections like List<T> use the Equals method when they're searching for items—like when you call the Contains or Remove method.

And finally, it **overloads the == and != operators** so you can use them to compare strings.

Sens-AI

Here's a **Sens-AI research challenge** for you. We modified the Card class from the Two Decks project in Chapter 8 to make it implement IEquatable and overload the == and != operators. Can you figure out why it works?

```
class Card(Values value, Suits suit) : IEquatable<Card>
{
    public Values Value => value;
    public Suits Suit => suit;
    public string Name => $"{Value} of {Suit}";
    public override string ToString() => Name;


    public bool Equals(Card? other) =>
         other != null && other.Value == Value && other.Suit == Suit;

    public override bool Equals(object? obj) => Equals(obj as Card);

    public override int GetHashCode() =>
         103237 * Value.GetHashCode() * Suit.GetHashCode();

    public static bool operator ==(Card? lhs, Card? rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
                return true;
           return false;
        }
        return lhs.Equals(rhs);
    }

    public static bool operator !=(Card? lhs, Card? rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
                return false;
            return true;
        }
        return !lhs.Equals(rhs);
    }
}
```

This is your opportunity to use the prompt engineering and AI research skills you've been working on. Looking for a starting point? Ask an AI to add comments to the code.

This code comes from an example on a Microsoft documentation page about value equality. Can you get the AI to find that page and summarize it for you?

Part of this Sens-AI challenge is using an AI chatbot to understand how value equality works in C# and .NET. Use the prompt engineering strategies and tactics that you worked on in Chapters 8 and 9 to learn about the concepts behind value equality, then test your knowledge by adding equality members to your Duck class from Chapter 8.

How do you define value equality for a class?

When a class or struct has **value equality**, it means that you can take two objects or values and compare them.

- What are the five guarantees of equivalence in C#? Why are they important for value equality?
- What is operator overloading? How do you overload operators in a class?
- Why does a class have to overrideEquals and GetHashCode when it implements IEquatable? How do you override GetHashCode? Why did we include the number 103237 in our version? *(Hint: it's a prime number.)*

★ *Challenge: Modify the Duck class to implement IEquatable<Duck>, override Object.Equals and GetHashCode, implement the == and != operators, and write a small app that tests them. Use operators to compare Duck objects. Add Duck objects to a List and make sure its Contains and Remove methods work.*

Records give your objects value equality automatically

There's an easier way to give your objects value equality. Instead of using a class or a struct, you can define a **record**, or a special kind of class that's specifically made for working with data. Use the `record` keyword and a primary constructor to define a record. Here's the record version of the Card class:

```
record Card(Values Value, Suits Suit)
{
    public string Name => $"{Value} of {Suit}";
    public override string ToString() => Name;
}
```

A record is like an ordinary class, but with extra equality superpowers. Try replacing the Card class in your Two Decks project from Chapter 8 with that Card record. Your app will still work the same.

Here's what a record can do for you:

- The parameters in the primary constructor are public, so you don't have to add properties to expose them. That's why we made the first letters of the Value and Suit parameters upper case.
- A record can make your code easier to read. A lot of classes use primary constructors or set `readonly` backing variables equal to parameters, and then add get properties to expose them. With a record, you don't need those extra getters, so it eliminates more boilerplate code.
- A record automatically implements value equality for you. You can use == or != to compare them, and if you add them to List<T> and other collections, the Contains and Remove methods work.
- A record automatically has a **ToString method** that returns a string with its public properties.
- A record is **immutable**, which means the fields in the primary constructor cannot change. That's one reason why they can be made public—there's no danger of modifying them.

Making an object immutable seems weird. Why would I ever want to prevent it from changing?

Making objects immutable is a great way to prevent bugs.

We've already seen how encapsulation and the `readonly` modifier can help you prevent bugs by keeping values from being accidentally overwritten. Records can help you to make sure entire objects are safe to use in the same way.

Cards are a great example of a kind of data that you never want to be able to mutate (or change its fields or other internal values). A playing card in the real world never changes its value—a Three of Clubs will never turn into a Jack of Diamonds. That makes cards a great candidate for records.

When you're designing the class hierarchy for an app, consider whether each class should be immutable.

Don't modify records—copy them

Remember the Duck class from Chapter 8? Here's a record that stores the same information:

```
record Duck(int Size = 24, KindOfDuck Kind = KindOfDuck.Mallard);

// Here's the KindOfDuck enum from Chapter 9
enum KindOfDuck { Mallard, Muscovy, Loon }
```

Did you notice how there are no **{** curly braces **}** after the declaration? You don't need to include them if you don't have any methods. You probably also noticed that we added **default values**.

Use the with keyword to create a modified copy of your Duck

Do this!

Create a new console app and add the Duck record declaration, KindOfDuck enum to it. Then add these lines to the top-level statements:

```
var duck = new Duck();
var smallerDuck = duck; // this won't work! we're just copying the reference
smallerDuck.Size = 26; // this line will have a syntax error
Console.WriteLine(smallerDuck);
```

That code has two problems. First, it uses = to try to copy the record, but that just copies the reference—smallerDuck still points to the same object. Second, it has this syntax error:

CS8852	Init-only property or indexer 'Duck.Size' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

Once you create a record, you can't set its properties because they're read-only—that's what makes it immutable. You *could* use the `new` keyword and set the individual primary constructor parameters:

```
var smallerDuck = new Duck(21, duck.Kind); // don't do this to copy records
```

But that's not a great solution for records. It's very common for a record's primary constructor to have many parameters in it. If you just need to change one, you can end up with code that's difficult to read.

That's where the **`with` keyword** comes in. Since records are immutable, they're meant to be copied. The `with` keyword gives you a convenient way to create a copy of your record while changing only specific parameters. Here's how it works:

```
var smallerDuck = duck with { Size = 21 };
```

When you use the = operator and the `with` keyword ***together***, they ***create a new instance of the record***—in this case, a new instance of the Duck record called `smallerDuck`, setting the Size parameter to `21` and all other parameters (in this case, Kind) to match `duck`.

Go ahead and **delete the two lines** that create smallerDuck and set its Size property, and **replace them** with the single line that uses the `with` keyword.

Now your app works—it copies the Duck instance and changes its size.

The "with" keyword lets you make a copy of a record and change its parameters so you don't have to write extra code to declare a whole new record.

there are no Dumb Questions

Q: Does a record live on the heap or the stack?

A: When you declared the Card record, you were declaring a class. Creating a new Card record creates a new object on the heap, and it acts just like any other object. You can even include the `class` keyword in your record declaration:

```
record class Card(Values Value, Suits Suit)
```

However, not all records are classes. A **record struct** is a record declared with the `struct` keyword:

```
record struct Card(Values Value, Suits
Suit)
```

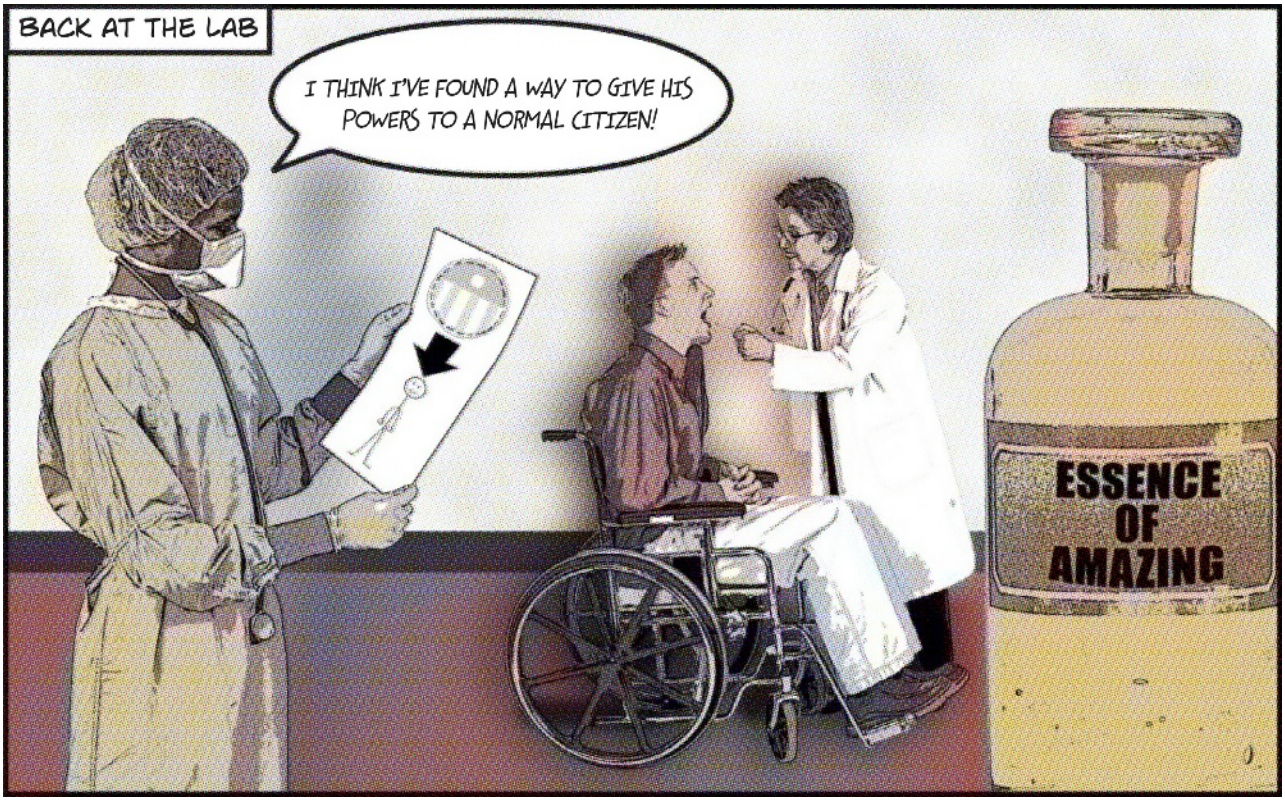
It works the same way as a record class: it's immutable, it implements equality, and you can use the `with` keyword to create new record structs—with one important exception: by default, a record struct is not immutable. However, adding the `readonly` keyword to the declaration causes the struct to be immutable.

Q: You said that records are more compact than other kinds of classes. What did you mean by that?

A: Records are more compact than other kinds of classes because the compiler automatically generates public properties for each of its parameters. Here's a Card class that uses a primary constructor, and expression-bodied members for its arguments:

```
public class Card(Values value, Suits suit)
{
  public Values Value => value;
    public Suits Suit => suit;
}
```

You've added similar expression-bodied members to several classes. Those lines are a good example of **boilerplate code**, or code that's repeated in lots of places with little variation.The record's primary constructor eliminates that boilerplate code. Since records give read-only access to the primary constructor parameters, you don't need to add separate properties for them, and you don't need a separate constructor that sets those properties. Add the equality implementation on top of that, and you get a lot of behavior in very few lines of code. It also automatically implements a ToString method. That's very compact!



there are no Dumb Questions

Q: OK, back up a minute. Why do I care about the stack?

A: Because understanding the difference between the stack and the heap helps you keep your reference types and value types straight. It's easy to forget that structs and objects work very differently—when you use the equals sign with both of them, they look really similar. Having some idea of how the CLR handles things under the hood helps you understand why reference and value types are different.

Q: And boxing? Why is that important to me?

A: Because you need to know when things end up on the stack, and you need to know when data's being copied back and forth. Boxing takes extra memory and more time. When you're only doing it a few times (or a few hundred times) in your program, then you won't notice the difference. But let's say you're writing a program that does the same thing over and over again, millions of times a second. That's not too far-fetched: your Unity games might do exactly that. If you find that your program's taking up more and more memory, or going slower and slower, then it's possible that you can make it more efficient by avoiding boxing in the part of the program that repeats.

Q: I get how you get a fresh copy of a struct when you set one struct variable equal to another one. But why is that useful to me?

A: One place that's really helpful is with encapsulation. Take a look at this code:

```
private Point location;
public Point Location {
  get { return location; }
}
```

If Point were a class, then this would be terrible encapsulation. It wouldn't matter that the location is private, because you made a public read-only property that returns a reference to it—so any other object would be able to access it.

Lucky for us, Point happens to be a struct. That means that the public Location property returns a fresh copy of the point. The object that uses it can do whatever it wants to that copy—none of those changes will make it to the private location field.

Q: How do I know whether to use a struct or a class?

A: Most of the time, programmers use classes. Structs have a lot of limitations that can really make it hard to work with them for large jobs. They don't support inheritance or abstraction, and only limited polymorphism, and you already know how important those things are for writing code.

Where structs come in really handy is if you have a small, limited type of data that you need to work with repeatedly. Unity vectors are good examples—some games will use them over and over again, possibly millions of times. Reusing a vector by assigning it to the same variable reuses that same memory on the stack. If Vector3 were a class, then the CLR would have to allocate new memory on the heap for each new Vector3, and it would constantly be garbage-collecting. So by making Vector3 a struct and not a class, the team that develops Unity gave you the gift of higher frame rates—without you having to do a thing.

A struct can be valuable for encapsulation because a read-only property that returns a struct always makes a fresh copy of it.

Sharpen your pencil

This method is supposed to kill an Evil`Clone` object by marking it for GC, but it doesn't work. Why not?

```
void SetCloneToNull(EvilClone clone) => clone = null;
```

……

……

Pool Puzzle

Your **job** is to take snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the code write the output shown below to the console when this app is executed.

Here's the output from the app.

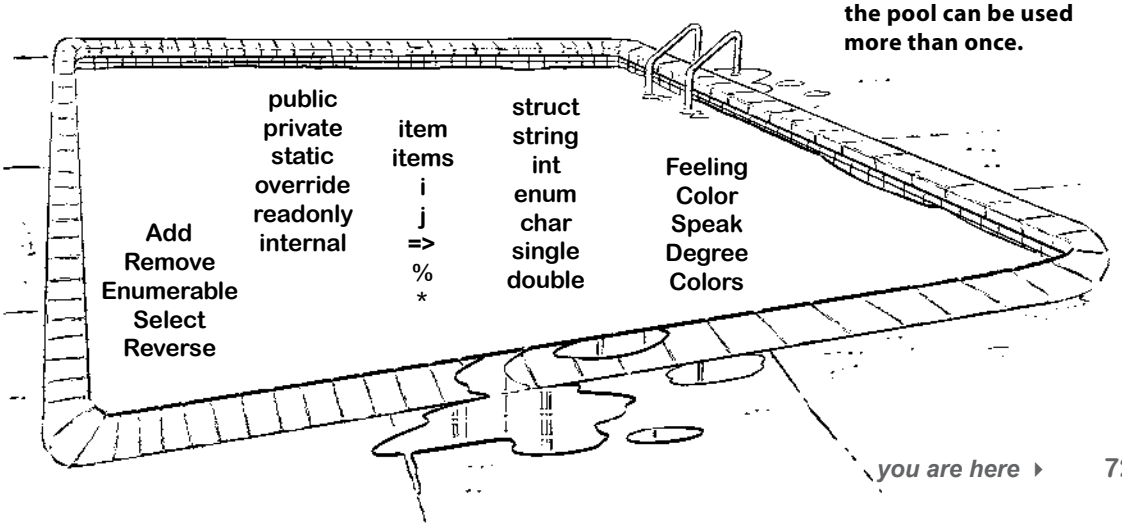
output

```
NO NEED TO OVERTHINK
```

```
var items = _____________.Range(0, 10)
    .Select((i) => new __________((__________)(i % 4), i))
    .ToList();

_____________.Range(0, 6)
    .__________((i) => new __________((__________)((i * 3) % 4), i * 3))
    .ToList()
    .ForEach((item) => items.Remove(item));

_____________.Range(0, 3)
    .___________()
    .__________((i) => new Feeling((Colors)i, 8 - i * 3))
    .ToList()
    .ForEach((item) => items.Remove(item));

for (int i = 0; i < 3; i++)
    _________[i] = _________[i] with { __________ = __________.Extra[2 - i] };

foreach (var item in _________)
    Console.Write($"{_________._________} ");

foreach (var item in _________)
    Console.Write(_________);

record Feeling(Colors Color, int Degree, string Speak = "")
{
    public __________ readonly _________[] Extra = ["TO", "NEED", "NO"];
    public override string ____________() => $"{Color}{(char)(68 + Degree)}";
}

_________ Colors { RT, OV, NO, IN }
```

The *LINQ ToList* method converts an IEnumerable to a List. The *List.ForEach method* uses a lambda to iterate through the list, calling the lambda for each item in the list. These are really useful methods! Can you figure out how they work?

Note: each snippet from the pool can be used more than once.



Pool Puzzle Solution

output

```
NO NEED TO OVERTHINK
```

The first LINQ query creates a List<Feeling> called items. The second one uses record equality to remove some items from the list.

```
var items = ___Enumerable___.Range(0, 10)
    .Select((i) => new ___Feeling___(( ___Colors___ )(i % 4), i))
    .ToList();

___Enumerable___.Range(0, 6)
    .___Select___((i) => new ___Feeling___(( ___Colors___ )((i * 3) % 4), i * 3))
    .ToList()
    .ForEach((item) => items.Remove(item));

___Enumerable___.Range(0, 3)
    .___Reverse___()
    .___Select___((i) => new Feeling((Colors)i, 8 - i * 3))
    .ToList()
    .ForEach((item) => items.Remove(item));

for (int i = 0; i < 3; i++)
    ___items___[i] = ___items___[i] with { ___Speak___ = ___Feeling___.Extra[2 - i] };

foreach (var item in ___items___)
    Console.Write($"{___item___.___Speak___} ");

foreach (var item in ___items___)
    Console.Write(___item___);

record Feeling(Colors Color, int Degree, string Speak = "")
{
    public ___static___ readonly ___string___[] Extra = ["TO", "NEED", "NO"];
    public override string ___ToString___() => $"{Color}{(char)(68 + Degree)}";
}

___enum___ Colors { RT, OV, NO, IN }
```

This loop uses the "with" keyword to update the Speak value in each of the records with a word from the static string array.

The Feeling record has three parameters (one with a default value), a static string array field, and a ToString method.

Sharpen your pencil Solution

This method is supposed to kill an EvilClone object by marking it for GC, but it doesn't work. Why not?

The clone parameter is just on the stack, so setting it to null doesn't do anything to the heap.

```
void SetCloneToNull(EvilClone clone) => clone = null;
```

All this method does is set its own parameter to null, but that parameter's just a reference to an EvilClone. It's like sticking a label on an object and peeling it off again.

Extension methods add new behavior to EXISTING classes

Remember the sealed modifier from Chapter 7? It's how you set up a class that can't be extended.

Sometimes you need to extend a class that you can't inherit from, like a sealed class (a lot of the .NET classes are sealed, so you can't inherit from them). C# gives you a flexible tool for that: **extension methods**. When you add a class with extension methods to your project, it **adds new methods that appear on classes** that already exist. All you have to do is create a static class, and add a static method that accepts an instance of the class as its first parameter using the `this` keyword.

So let's say you've got a **sealed** OrdinaryHuman class (so you can't extend it with inheritance):

```
sealed class OrdinaryHuman(int weight) {
    private int age;
    int weight;

    public void GoToWork() { /* code to go to work */ }
    public void PayBills() { /* code to pay bills */ }
}
```

The OrdinaryHuman class is sealed, so it can't be subclassed. But what if we want to add a method to it?

You use an extension method by adding a static method that specifies the type to extend in the first parameter using the **`this` keyword**. We want to add a method that extends the OrdinaryHuman class. To do that, we add a **static method** that uses the `this` keyword in the first parameter, like this: `this OrdinaryHuman h`

The AmazeballsSerum class adds the BreakWalls extension method to the OrdinaryHuman class:

```
static class AmazeballsSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return ($"I broke through a wall of {wallDensity} density.");
    }
}
```

Extension methods are always static methods, and they have to live in static classes.

As soon as the AmazeballsSerum class is added to the project, OrdinaryHuman gets a BreakWalls method. So now your top-level statements can use it:

```
OrdinaryHuman steve = new OrdinaryHuman(185);
Console.WriteLine(steve.BreakWalls(89.2));
```

Run the app—it prints this output:

```
I broke through a wall of 89.2 density.
```

And that's it! All you need to do is add the AmazeballsSerum class to your project, and suddenly every OrdinaryHuman class gets a brand-new BreakWalls method.

When the program creates an instance of the OrdinaryHuman class, it can access the BreakWalls method directly—as long as the AmazeballsSerum class is in the project. Go ahead, try it out! Create a new console application and add the two classes and the top-level statements to it. Debug into the BreakWalls method and see what's going on.

The LINQ methods you learned about in Chapter 9 are extension methods for the type IEnumerable<T>. Use the IDE's "Go to definition" to see how they're declared.

Extending a fundamental type: string

Let's explore how extension methods work by extending the String class.
Create a new Console App project, and add a file called *HumanExtensions.cs*.

Do this!

1 **Put all of your extension methods in a separate namespace.**
Add a new file for a class called *ExtendAHuman.cs*. Surround it with a namespace declaration, and mark it public and static:

```
namespace AmazingExtensions {
  public static class ExtendAHuman {
```

The class needs to be static because it contains extension methods, and public because you'll be using it from another namespace. We used a block-scoped namespace to make the code easier to read.

2 **Create the static extension method, and define its first parameter as `this` and then the type you're extending.**
The two main things you need to know when you declare an extension method are that the method needs to be static and **it takes the class it's extending as its first parameter**:

The extension method must be static too.

```
    public static bool IsDistressCall(this string s) {
```

"this string s" says we're extending the String class and uses the parameter s to access the string being used to call the method.

3 **Finish the extension method.**
This method checks the string to see if it contains the word "Help!"—if it does, then that string is a distress call, which every superhero is sworn to answer:

```
      if (s.Contains("Help!"))
        return true;
      else
        return false;
    }
  }
}
```

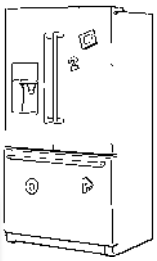
This uses the String.Contains method to see if the string contains the word "Help!"—and that's definitely not something an ordinary string normally does.

4 **Use your new IsDistressCall extension method.**
Add `using AmazingExtensions;` to the top of the file with your Program class. Then add code to the class that creates a string and calls its IsDistressCall method. You'll see your extension in the IntelliSense window:

```
1  using AmazingExtensions;
2
3  string message = "Evil clones are wreaking havoc. Help!";
4  message.IsDistressCall
```

IsDistressCall (extension) bool string.IsDistressCall()

As soon as you add the using directive to add the namespace with your static class, the IntelliSense window contains your new extension method.



Extension Magnets

Arrange the magnets to produce this output:

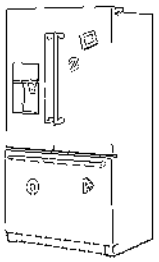
a buck begets more bucks

```
namespace Upside
{
```

Extension methods are on this side.

Top-level statements are on this side.

```
}
string s = i.ToPrice();
public static string ToPrice
i = 3;
(b == true)
    return "be";
int i = 1;
else
    return " more bucks";
(this int n) {
if (n == 1)
    return "a buck ";
}
}
}
(this bool
.SendIt();
public static string Green
public static void SendIt
}
using Upside;
b = false;
b.Green().SendIt();
Console.Write(s);
b.Green().SendIt();
s.SendIt();
bool b = true;
public static class Margin {
(this string s) {
i.ToPrice()
else
    return "gets";
```



Extension Magnets Solution

Arrange the magnets to produce this output:

a buck begets more bucks

The Upside namespace has the extensions. The Sideways namespace has the entry point.

Extension methods are on this side.

The Margin class extends a string by adding a method called SendIt that just writes the string to the console, and it extends int by adding a method called ToPrice that returns "a buck" if the int's equal to 1, or "more bucks" if it's not.

Top-level statements are on this side.

```
namespace Upside {
  public static class Margin {
    public static void SendIt (this string s) {
        Console.
      Write(s);
    }

    public static string ToPrice (this int n) {
      if (n == 1)
          return "a buck ";
      else
          return " more bucks";
    }

    public static string Green (this bool b) {
      if (b == true)
          return "be";
      else
          return "gets";
    }
  }
}
```

The Green method extends a bool—it returns the string "be" if the bool is true, and "gets" if it's false.

```
using Upside;

int i = 1;
string s = i.ToPrice();
s.SendIt();
bool b = true;
b.Green().SendIt();
b = false;
b.Green().SendIt();
i = 3;
i.ToPrice() .SendIt();
```

The top-level statements call the SendIt, ToPrice, and Green extension methods.

Exercise

It's time for Captain Amazing to defeat the Swindler's army of clones…but he needs your help! Can you build a game to help use his Captain Amazing Clone Targeting System™ to stop the Swindler's plan and save Objectville?

```
Name: ZapBot3000        Smasho  KillMeister     PREDATRON       Larry
ID:     1       2       3       >4<     5       6

Target clones: Name = up/down arrows, ID = left/right arrows, enter = fire

Clones: 17          Time: 35.3s
[ZapBot3000 #4] [Smasho #2]     [KillMeister #3]        [Predatron #1]  [Larry #1]
[Larry #2]      [KillMeister #4]        [ZapBot3000 #1] [Smasho #4]     [Smasho #6]
[KillMeister #2]        [Larry #3]      [ZapBot3000 #3] [Smasho #6]     [Smasho #3]
[Smasho #6]     [KillMeister #5]        |
```

This is a real-time game. When you start it, click on the window to focus on it so it can read the keystrokes.

- The goal of the game is to eliminate all of the clones. Each clone has one of five names and a random ID number from 1 to 6. The game starts with five clones, one for each of the five names.
- The bottom of the display lists the clone count, elapsed time, and all of the clones. There can be up to 30 clones.
- Every 1.5 seconds a new clone of one of the existing clones is added with the same name and a new random ID. *(Hint: choose a random Clone from the list and use the* `with` *keyword to create a new copy of it.)*
- The targeting system is at the top of the screen. The player selects a name using the up and down arrow keys and an ID using the left and right arrow keys. The selected name is in UPPERCASE, the selected ID has >carets<.
- Pressing the Enter key causes every clone with a matching name and ID to be removed. When the last clone is removed, the game displays this message: `"You won in XX.X seconds."`

Start with the Clone record

The Clone class has two public properties to hold the name and ID. It has constants to define a maximum ID number, the number of seconds between adding clones, and the frame delay.

```
record Clone(string Name, int Id)
{
    public const int MAX_ID = 6;
    public const double ADD_SECONDS = 1.5;
    public const int FRAME_DELAY_MILLISECONDS = 25;

    public static IReadOnlyList<string> CloneNames =
        [ "ZapBot3000", "Smasho", "KillMeister", "Predatron", "Larry" ];

    public override string ToString() => $"[{Name} #{Id}]";
}
```

Each clone never changes its name or ID, and you'll need to compare clones to find the ones that the player targeted, so a record makes perfect sense for this.

You'll need some new tools to make this game work—can AI help you figure out how to use them?

One of the most important skills a developer can learn is figuring out new tools to help build apps. Here are a few useful tools that you'll use in the code. Get some research practice by using AI to learn more about them.

- You learned about frames in the Unity labs. Start each frame by calling **Console.Clear** to clear the screen. Write the frame text, and call `Thread.Sleep(Clone.FRAME_DELAY_MILLISECONDS)` to delay until the next frame.
- You've used Console.ReadKey to read a single key that the user pressed. Normally that pauses the app and waits for the user to press a key, but we want the game to keep running if the user hasn't pressed one. You can keep the game running by checking the **Console.KeyAvailable** property—it returns true if the user pressed a key.
- Console.ReadKey().Key returns a **ConsoleKey** enum value. Use its UpArrow, DownArrow, LeftArrow, RightArrow, and Enter values to figure out which key the user pressed. This works really well in a `switch` statement.
- The **DateTime struct** lets you work with times. **DateTime.Now** returns the current time. If you store the last time a clone was added in a variable called lastAddTime, `(DateTime.Now - lastAddTime).TotalSeconds` will return the number of seconds that have elapsed since then as a double.
- The list of clones shows five clones per line. Create a LineBreakOrTab lambda expression so you can write the clones like this: `clones.ForEach(clone => Console.Write($"{clone}{LineBreakOrTab()}"));`

Exercise Solution

We used a Select LINQ query to create our initial list of Clone objects.

```
var clones = Clone.CloneNames
    .Select(name => new Clone(name, Random.Shared.Next(1, Clone.MAX_ID))).ToList();

int selectedName = 0;
int selectedId = 1;
var startTime = DateTime.Now;
var lastAddTime = DateTime.Now;

while (clones.Count > 0)
{
    Console.Clear();
    Console.Write("Name: ");
    for (var i = 0; i < Clone.CloneNames.Count; i++)
    {
        var name = Clone.CloneNames[i];
        Console.Write($"{(i == selectedName ? name.ToUpper() : name)}\t");
    }
    Console.Write($"{Environment.NewLine}ID:  \t");
    for (int i = 1; i <= Clone.MAX_ID; i++)
        Console.Write($"{(i == selectedId ? $">{i}<" : i)}\t");

    Console.WriteLine($@"

Target clones: Name = up/down arrows, ID = left/right arrows, enter = fire

Clones: {clones.Count}          Time: {(DateTime.Now - startTime).TotalSeconds:0.0}s");

    int count = 0;
    string LineBreakOrTab() => ++count % 5 == 0 ? Environment.NewLine : "\t";
    clones.ForEach(clone => Console.Write($"{clone}{LineBreakOrTab()}"));

    if (Console.KeyAvailable)
    {
        switch (Console.ReadKey().Key)
        {
            case ConsoleKey.UpArrow:
                if (selectedName > 0) selectedName--;
                break;
            case ConsoleKey.DownArrow:
                if (selectedName < Clone.CloneNames.Count - 1) selectedName++;
                break;
            case ConsoleKey.LeftArrow:
                if (selectedId > 1) selectedId--;
                break;
            case ConsoleKey.RightArrow:
                if (selectedId < Clone.MAX_ID) selectedId++;
                break;
            case ConsoleKey.Enter:
                var target = new Clone(Clone.CloneNames[selectedName], selectedId);
                clones.Remove(target);
                break;
        }
    }

    if ((DateTime.Now - lastAddTime).TotalSeconds > Clone.ADD_SECONDS && clones.Count < 99)
    {
        var clone = clones[Random.Shared.Next(clones.Count)];
        clones.Add(clone with { Id = Random.Shared.Next(1, Clone.MAX_ID + 1) });
        lastAddTime = DateTime.Now;
    }
    Thread.Sleep(Clone.FRAME_DELAY_MILLISECONDS);
}

Console.WriteLine($"You won in {(DateTime.Now - startTime).TotalSeconds:0.0} seconds");
```

Is your app flickering? Replace Clear() with SetCursorPosition(0, 0), which moves your cursor without clearing the screen, and add a call to Console.Clear() at the top of the code. You'll see extra characters that don't get deleted. Can you fix them so the app looks good?

The game needs to keep track of two times: the time the game started and the time the last clone was added.

Use selectedName and selectedId to display the targeted name in all caps and ID with >carets<.

The LineBreakOrTab lambda expression uses ++ and the ? : ternary operator to return four tabs and then a line break.

If there's a key available, this switch statement reads it and updates the target selection or removes the target clone.

Since Clone is a record, you can create a new Clone object and use the List.Remove method to remove any matches.

This chooses a random clone and uses the "with" keyword to make a copy with the same name but a new ID.

there are no Dumb Questions

Q: Tell me again why I wouldn't add the new methods I need directly to my class code, instead of using extensions?

A: You could do that, and you probably should if you're just talking about adding a method to one class. Extension methods should be used pretty sparingly, and only in cases where you absolutely can't change the class you're working with for some reason (like it's part of the .NET Framework or another third party). Where extension methods really become powerful is when you need to extend the behavior of something you ***wouldn't normally have access to***, like a type or an object that comes for free with the .NET Framework or another library.

One more point to remember about extension methods: you don't gain access to any of the class's internals by creating an extension method, so it's still acting as an outsider. →

Q: Why use extension methods at all? Why not just extend the class with inheritance?

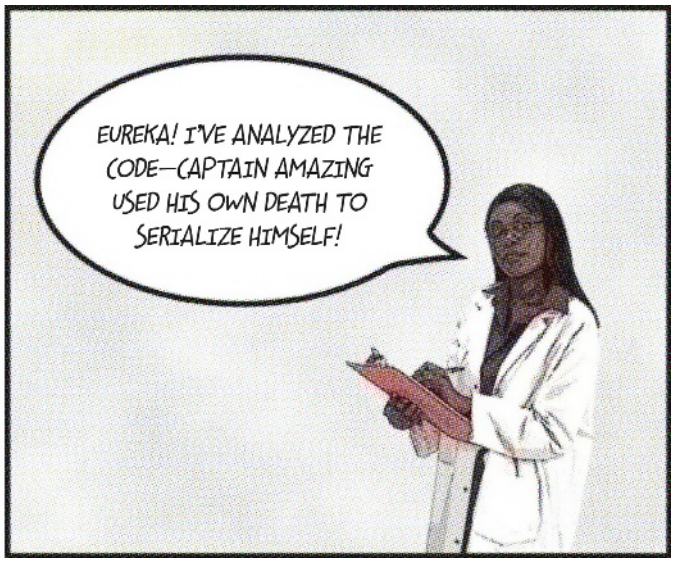
A: If you can extend the class, then you'll usually end up doing that—extension methods aren't meant to be a replacement for inheritance. But they come in really handy when you've got classes that you can't extend. With extension methods, you can change the behavior of whole groups of objects, and even add functionality to some of the most basic classes in the .NET Framework. Extending a class gives you new behavior, but requires that you use the new subclass if you want to use that new behavior.

Q: Does my extension method affect all instances of a class or just a certain instance of the class?

A: It will affect all instances of a class that you extend. In fact, once you've created an extension method, the new method will show up in your IDE alongside the extended class's normal methods.

Bullet Points

- A **nullable value type** can either have a value or be set to null. When T is a value type, T? is an alias for Nullable<T>, a struct that has properties that help to handle potentially null values.
- The **null-coalescing operator `??`** checks for nulls automatically and returns an alternative. The **`??=` operator** assigns to a variable only if it's null.
- Any reference type can use == and work with collection methods that use comparison by implementing **value equality**.
- A **record** is a class that automatically has value equality. It is immutable, which means it can't be modified, using a primary constructor with parameters that are read-only.
- The parameters in a record's **primary constructor** are public, so you don't have to add properties for them.
- A record automatically has a **ToString method** that returns a string with its public properties.
- A record is a **class** by default, and you can include the `class` keyword in the declaration.
- Use the = operator and the **`with` keyword** to copy a record instance and change some of its properties during the copy operation.
- You can declare a **record struct**, which is a value type. It works the same way as a record, except that it is not immutable. Adding the `readonly` keyword to the record struct declaration makes it immutable.
- **Boilerplate code** is code that's repeated in lots of places with little variation.
- Records help to **eliminate boilerplate code** by automatically exposing their public properties, eliminating the need for backing fields, and providing a ToString method.
- **Extension methods** add new behavior to existing classes, including sealed classes that can't be subclassed.
- Create an extension method by declaring it as static and using the **`this` keyword** in its first parameter to specify the type being extended. The method can be used from any reference to that class.



The UNIVERSE

CAPTAIN AMAZING REBORN

DEATH WAS NOT THE END

by Balki Burns
UNIVERSE STAFF WRITER

OBJECTVILLE

Captain Amazing deserializes himself, making a stunning comeback.

In a stunning turn of events, Captain Amazing has returned to Objectville. Last month, Captain Amazing's coffin was found empty, and only a strange note left where his body should have been. Analysis of the note revealed Captain Amazing's object DNA—all his last fields and values, captured faithfully in binary format.

Today, that data has sprung to life. The Captain is back, deserialized from his own brilliant note. When asked how he conceived of such a plan, the Captain merely shrugged and mumbled, "Chapter 11." Sources close to the Captain refused to comment on the meaning of his cryptic reply, but did admit that prior to his failed assault on the Swindler, the Captain had spent a lot of time reading books, studying Dispose methods and persistence. We expect Captain Amazing…

Captain Amazing is back!

…see AMAZING on A-5

12 exception handling

Putting out fires gets old

When you have to deal with error after error, it's called "putting out fires."

Imagine: it's a few years from now. You spent all that time working on your C# skills and continuing to learn and improve, and now you're one of the most senior developers at a big tech company. Now you're getting panicked phone calls in the middle of the night from your support team because **your app crashes**, or **doesn't behave like it's supposed to**. You want to spend your time writing code, not putting out fires! Because nothing pulls you out of the programming groove like having to fix a strange bug you don't recognize. Luckily, C# has **exception handling**, which lets you write code to **deal with problems** that come up. Better yet, you can even plan for those problems, and **keep things running** when they happen.

Your hex dumper reads a filename from the command line

At the end of Chapter 10, you built a hex dumper that uses command-line arguments to dump any file. You used the project properties in the IDE to set the arguments for the debugger, and you saw how to call it from a Windows command prompt or macOS Terminal window.

```
C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0>HexDump binarydata.dat
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6 ?)è??Hello!/??tö
0010: d8 f5 43 45             --                         OoCE
```

```
publish — -zsh — 79×5
MacBook-Pro publish % ./HexDump binarydata.dat
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6    .)...Hello!/..t.
0010: d8 f5 43 45             --                            ..CE
MacBook-Pro publish %
```

But what happens if you give HexDump an invalid filename?

When you modified your HexDump app to use command-line arguments, we asked you to be careful to specify a valid filename. What happens when you give it an invalid filename? Try running your app again from the command line, but this time give it the argument `invalid-filename`. Now it ***throws an exception***.

The exception has a class name and a message...

```
C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0>HexDump invalid-filename
Unhandled exception. System.IO.FileNotFoundException: Could not find file 'C:\Users\Public\source\repos\HexDump\HexDump\
bin\Debug\net8.0\invalid-filename'.
File name: 'C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0\invalid-filename'
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath, FileMode mode, FileAccess access, FileShare
 share, FileOptions options)
   at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(String fullPath, FileMode mode, FileAccess access, FileShare share
, FileOptions options, Int64 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.Strategies.OSFileStreamStrategy..ctor(String path, FileMode mode, FileAccess access, FileShare share, Fi
leOptions options, Int64 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String path, FileMode mode, FileAccess access, FileShare
 share, FileOptions options, Int64 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.File.OpenRead(String path)
   at Program.<Main>$(String[] args) in C:\Users\Public\source\repos\HexDump\HexDump\Program.cs:line 18
```

...and a stack trace.

Use the project settings to set the program's argument to an invalid filename and run the app in the IDE's debugger. Now you'll see it throw an exception with the same class name (System.IO.FileNotFoundException) and a similar "Could not find file" message.

An unhandled exception means our app encountered a problem we didn't account for.

```
Exception Unhandled

System.IO.FileNotFoundException: 'Could not find file 'C:\Users
\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0\invalid-
filename'.'

Show Call Stack | View Details | Copy Details | Start Live Share session
▸ Exception Settings
```

The IDE halts the debugger on the line that threw the exception and shows you information in its *Exception Unhandled window*. You can even see the call stack in its Call Stack window.

Exercise

This code is broken. The code throws five different exceptions, and the error messages you'll see in the IDE or the console are on the right. It's your job to **match the line of code** that has a problem with the **exception that line generates**. Read the exception messages for hints. Keep in mind that ***this code does not work***. If you add the top-level statements and HoneyBee class that follow to an app and run them, it will halt after the first exception is thrown. Just read through the code and match each exception to the line that *would* throw it if it *could* run.

```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;

HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.Name);

double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int)totalHoney; i++)
{
    beesWeCanFeed += i.ToString();
}
int numberOfBees = int.Parse(beesWeCanFeed);

int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;

anotherBee = null;
if (dronesPerQueen < 10)
{
    anotherBee.Capacity = 12.6;
}

class HoneyBee(double capacity, string name)
{
    public double Capacity
    {
       get => capacity;
       set => capacity = value;
    }
    public string Name
    {
        get => name;
        set => name = value;
    }
}
```

The double.Parse method converts a string to a double, so if you pass it a string (such as "32.7"), it will return the equivalent double value (32.7D). What do you think happens if you pass it a string that can't be converted to a double?

System.OverflowException: 'Value was either too large or too small for an Int32.' (1)

System.NullReferenceException: 'Object reference not set to an instance of an object.' (2)

System.InvalidCastException: 'Unable to cast object of type 'HoneyBee' to type 'System.Single'.' (3)

System.DivideByZeroException: 'Attempted to divide by zero.' (4)

System.FormatException: 'The input string 'Buzzy' was not in a correct format.' (5)

Exercise Solution

Your job was to match each line of code that has a problem with the exception that line generates.

```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;
```

The code to cast myBee to a float compiles, but there's no way to convert a HoneyBee object to a float value. When your code runs, the CLR has no idea how to actually do that cast, so it throws an InvalidCastException.

System.InvalidCastException: 'Unable to cast object of type 'HoneyBee' to type 'System.Single'.' (3)

IDE Tip: Set Next Statement/Jump to Cursor

You can reproduce these exceptions in the IDE by pasting your code in and running it. Place a breakpoint on the first line of code, then right-click on the line of code you want to run and choose **Set Next Statement** (Visual Studio) or **Jump to Cursor** (VSCode). When you continue running, your app will jump straight to that statement.

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.Name);
```

System.FormatException: 'The input string 'Buzzy' was not in a correct format.' (5)

The Parse method wants you to give it a string in a certain format. "Buzzy" isn't a string it knows how to convert to a number. That's why it throws a FormatException.

```
double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int)totalHoney; i++)
{
    beesWeCanFeed += i.ToString();
}
int numberOfBees = int.Parse(beesWeCanFeed);
```

The for loop will create a string called beesWeCanFeed that contains a number with over 60 digits in it. There's no way an int can hold a number that big, and trying to cram it into an int will throw an OverflowException.

System.OverflowException: 'Value was either too large or too small for an Int32.' (1)

You'd never actually get all these exceptions in a row—the program would throw the first exception and then halt. You'd only get to the second exception if you fixed the first.

Exercise Solution

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

It's really easy to throw a DivideByZeroException. Just divide any number by zero.

System.DivideByZeroException: 'Attempted to divide by zero.' 4

Dividing any integer by zero always throws this kind of exception. Even if you don't know the value of queens, you can prevent it just by checking the value to make sure it's not zero **before** you divide it into drones.

I can see just from looking at that code that it's trying to divide by zero. I bet that exception ***didn't have to happen.***

That DivideByZero error didn't have to happen.

You can see just by looking at the code that there's something wrong. If you think about it, the same goes for the other exceptions—this whole Exercise was about spotting those exceptions without running the code. Every one of those exceptions was ***preventable***. The more you know about exceptions, the better you'll be at keeping your apps from crashing.

```
anotherBee = null;
if (dronesPerQueen < 10)
{
    anotherBee.Capacity = 12.6;
}
```

Setting the anotherBee reference variable equal to null tells C# that it doesn't point to anything. So instead of pointing to an object, it points to nothing. Throwing a NullReferenceException is C#'s way of telling you that there's no object whose DoMyJob method can be called.

System.NullReferenceException: 'Object reference not set to an instance of an object.' 2

When your program throws an exception, the CLR generates an Exception object

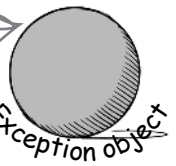
You've been looking at the CLR's way of telling you something went wrong in your program: an **exception**. When an exception occurs in your code, an object is created to represent the problem. It's called—no surprise here—Exception.

For example, suppose you have an array with four items, and you try to access the 16th item (index 15, since we're zero-based here):

```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```

This code is obviously going to cause problems.

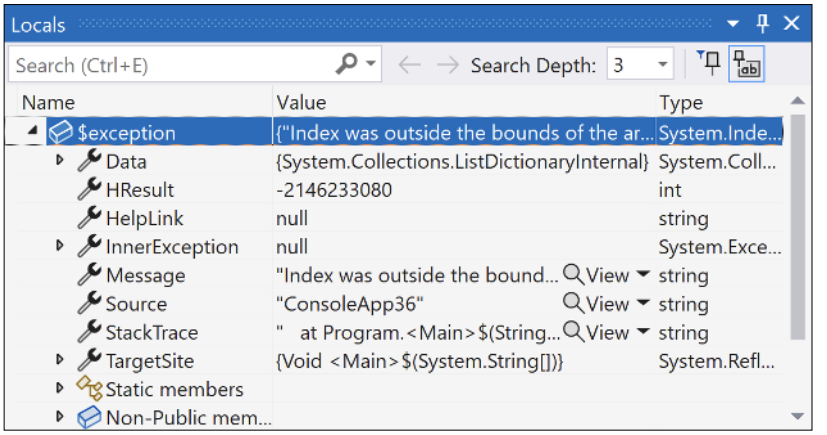
ex-cep-tion, noun. a person or thing that is excluded from a general statement or does not follow a rule. *While Jamie usually hates peanut butter, they made an* ***exception*** *for Parker's peanut butter fudge.*



As soon as your program runs into an exception, it generates an object with all the data it has about the problem.

When the IDE halts because the code threw an exception, you can see the details of the exception by **expanding `$exception` in the Locals window**. The Locals window shows you all of the variables currently ***in scope*** (which means the current statement has access to them).

The Exception object has a message that tells you what's wrong, and a stack trace, or a list of all of the calls that were made leading up to the statement that caused the exception.



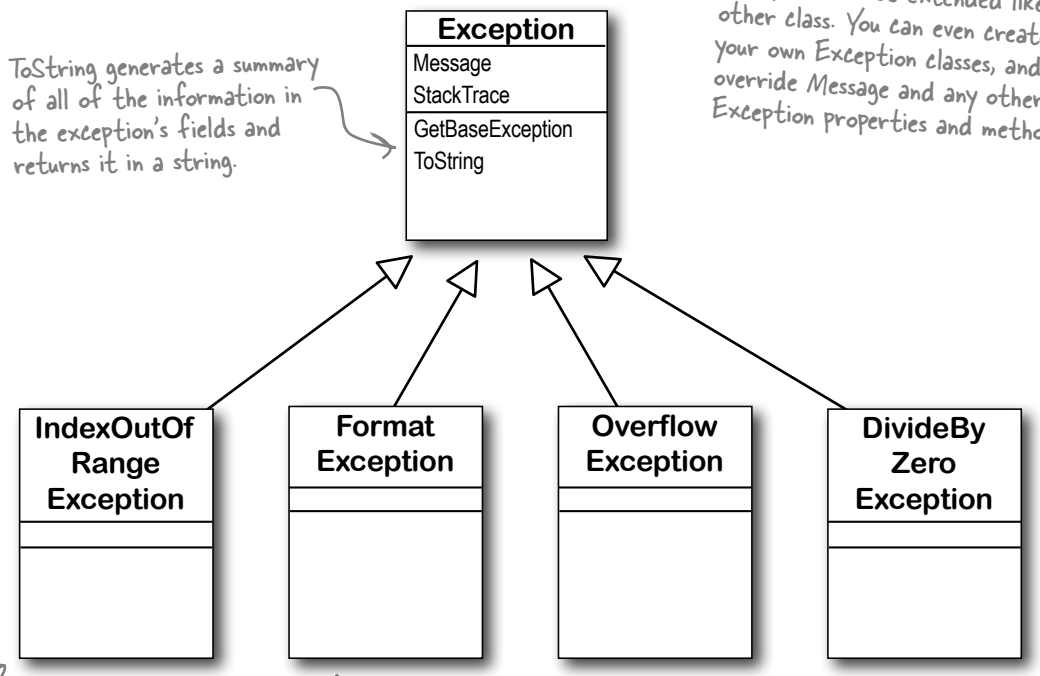
The CLR goes to the trouble of creating an object because it wants to give you all the information it has about what caused the exception. You may have code to fix, or you may just need to make some changes to how you handle a particular situation in your program.

This particular exception is an **IndexOutOfRangeException**, which gives you an idea of what the problem is: you're trying to access an index in the array that's out of range. You've also got information about exactly where in the code the problem occurred, making it easier to track down and solve (even if you've got thousands of lines of code).

All Exception objects inherit from System.Exception

.NET has lots of different exceptions it may need to report. Since many of these have a lot of similar features, inheritance comes into play. .NET defines a base class, called Exception, that all specific exception types inherit from.

The Exception class has a couple of useful members. The Message property stores an easy-to-read message about what went wrong. StackTrace tells you what code was being executed when the exception occurred, and what led up to the exception. (There are others, too, but we'll use those first.)



ToString generates a summary of all of the information in the exception's fields and returns it in a string.

Exception can be extended like any other class. You can even create your own Exception classes, and override Message and any other Exception properties and methods.

It's really useful that .NET gives us so many types of exceptions, because each different exception is thrown for a different situation. You can learn a lot about the unexpected action that's causing an exception just by looking at which one was thrown.

Brain Power

Your HexDump app throws an exception if you pass it an invalid filename on the command line. What can you do to deal with that problem?

Sleuth it Out

The first step in troubleshooting any exception is taking a really close look at all of the information that it gives you. When you're running a console app, that information is written to the console. Let's take a closer look at the diagnostic information our app printed (we moved our app to the C:\HexDump folder to make the paths in the stack trace shorter):

```
C:\HexDump\bin\Debug\net8.0> HexDump invalid-filename
Unhandled exception. System.IO.FileNotFoundException: Could not find file 'C:\HexDump\bin\Debug\
net8.0\invalid-filename'.
File name: 'C:\HexDump\bin\Debug\net8.0\invalid-filename'
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   ...
   at Program.<Main>$(String[] args) in C:\HexDump\Program.cs:line 12
```

Here's what we noticed:

- The exception class: `System.IO.FileNotFoundException`
- The exception message: `Could not find file 'C:\HexDump\bin\Debug\net8.0\invalid-filename'.`
- Additional diagnostic information: `File name: 'C:\HexDump\bin\Debug\net8.0\invalid-filename'`
- The first five lines in the stack trace come from classes in the System.IO namespace.
- The last line of the stack trace is in our namespace, HexDump—and it includes a line number. Here's what's on that line: `using (Stream input = File.OpenRead(args[0]))`.

Reproduce the exception in the debugger

At the end of Chapter 10, we showed you how to set the application arguments when you run an app in the debugger. Set the arguments to `invalid-filename`, then place a breakpoint on the line in the app that's throwing the exception. Run the app, then when you get to the breakpoint, step over that statement and see an exception in the IDE.

Add code to <u>prevent</u> the exception

The app is throwing an exception because it's trying to read a file that doesn't exist. So let's ***prevent*** this exception from happening by checking if the file exists first. If it doesn't, instead of using File.OpenRead to open a stream with the contents of the file, we'll use Console.OpenStandardInput, which returns a stream of your app's **standard input** (or **stdin**). Start by **adding this GetInputStream method** to your app:

```
Stream GetInputStream(string[] args) {
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
        return File.OpenRead(args[0]);
}
```

Console.OpenStandardInput **returns a Stream object connected to the app's standard input. If you pipe input into the app or run it in the IDE and type `console` or `terminal`, anything you type or pipe in ends up in the stream.**

Then modify the line of the app that throws the exception to call the new method:

```
        using (Stream input = GetInputStream(args))
```

Now run your app in the IDE. It doesn't throw an exception. Instead, it's reading from standard input. Test it out:

- Type some data and press Enter—it will show a hex dump of everything you typed, ending with a return (`0d 0a` on Windows, `0a` on Mac). The stream only adds data *after you press Enter*. Press ^Z (Windows) or ^D (Mac) to end.
- Run your app from the command line: `HexDump << input.txt` (or `./HexDump << input.txt` on a Mac). The app will pipe data from *input.txt* into the stdin stream and dump all of the bytes in the file.

there are no Dumb Questions

Q: What *is* an exception, really?

A: It's an object that the CLR creates when there's a problem. You can specifically generate exceptions in your code too—in fact, you've done that already using the `throw` keyword.

Q: An exception is an *object*?

A: Yes, an exception is an object. The CLR generates it to give you as much information as it can about exactly what was going on when it executed the statement that threw the exception. Its properties—which you saw when you expanded `$exception` in the Locals window—tell you information about the exception. For example, its Message property will have a useful string like *Attempted to divide by zero* or *Value was either too large or too small for an Int.32*.

Q: Why are there so many types of Exception objects?

A: Because there are so many ways that your code can act in unexpected ways. There are a lot of situations that will cause your code to simply crash. It would be really hard to troubleshoot the problems if you didn't know why the crash happened. By throwing different kinds of exceptions under different circumstances, the CLR is giving you a lot of really valuable information to help you track down and correct the problem.

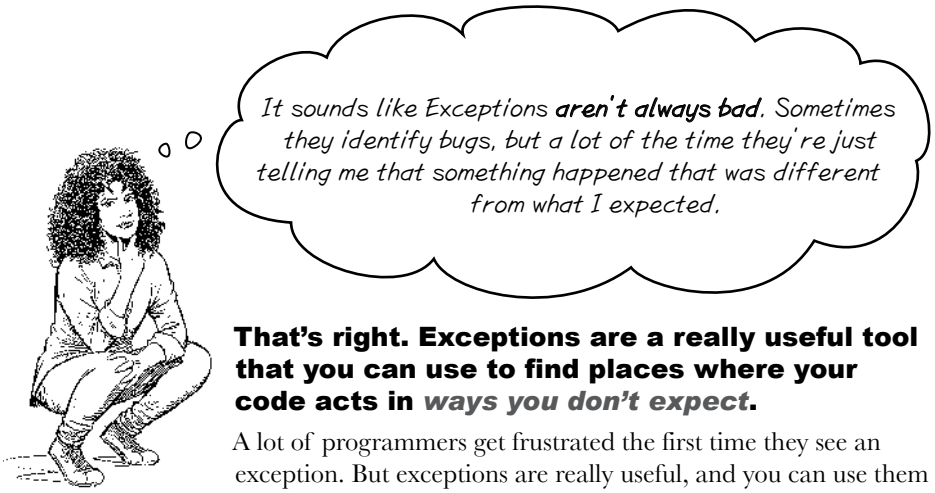
Q: Does that mean that when my code throws an exception, it's not necessarily because I did something wrong?

A: Exactly. Sometimes your data's different than you expected it to be, like in the example where we had a statement working with an array that was a lot shorter than anticipated. Also, don't forget that real, live, actual human beings are using your program, and they'll often act in unpredictable ways. Exceptions are C#'s way to help you handle those unexpected situations so that your code still runs smoothly and doesn't simply crash or give a cryptic, useless error message.

Q: So exceptions are there to help me, not just cause me grief and force me to track down frustrating bugs at 3:00 AM?

A: Yes! Exceptions are all about helping you expect the unexpected. A lot of people get frustrated when they see code throw an exception. Instead, think about them as C#'s way of helping you track down problems and debug your programs.

Exceptions are all about helping you find and fix situations where your code behaves in ways you didn't expect.



That's right. Exceptions are a really useful tool that you can use to find places where your code acts in *ways you don't expect*.

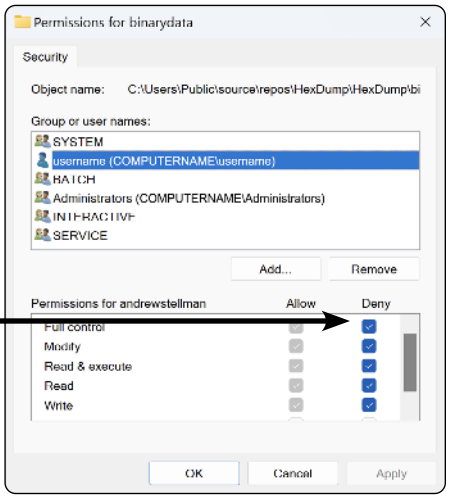
A lot of programmers get frustrated the first time they see an exception. But exceptions are really useful, and you can use them to your advantage. When you see an exception, it gives you a lot of clues to help you figure out why your code is reacting in a way that you didn't anticipate. That's good for you: it lets you know about a new scenario that your program has to handle, and it gives you an opportunity to ***do something about it***.

There are some files you just can't dump

In Chapter 9, we talked about making your code **robust** so that it can deal with bad data, malformed input, user errors, and other unexpected situations. Dumping stdin if no file is passed on the command line or if the file doesn't exist is a great start to making the hex dumper robust.

But are there still some cases that we need to handle? For example, what if the file exists but it's not readable? Let's see what happens if we remove read access from a file, and then try to read it:

- ***On Windows:*** Right-click the file in the Windows Explorer, go to the Security tab, and click Edit to change the permissions. Select **your user name** and check all of the **Deny boxes**.
- ***On a Mac:*** In a Terminal window, change to the folder with the file you want to dump and run this command (replace `binarydata.dat` with the name of your file): `chmod 000 binarydata.dat`.



Now that you've removed read permissions from your file, try running your app again, either in the IDE or from the command line.

You'll see an exception—the stack trace shows that the **`using` statement called the GetInputStream method**, which eventually caused the FileStream to throw a System.UnauthorizedAccessException:

```
C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0>hexdump binarydata.dat
Unhandled exception. System.UnauthorizedAccessException: Access to the path 'C:\Users\
Public\source\repos\HexDump\HexDump\bin\Debug\net8.0\binarydata.dat' is denied.
   ...
   at HexDump.Program.Main(String[] args) in C:\HexDump\Program.cs:line 20
```

*Wait a second. Of course the program's gonna crash. I gave it an unreadable file. Users screw up all the time. You can't expect me to do anything about that...**right?***

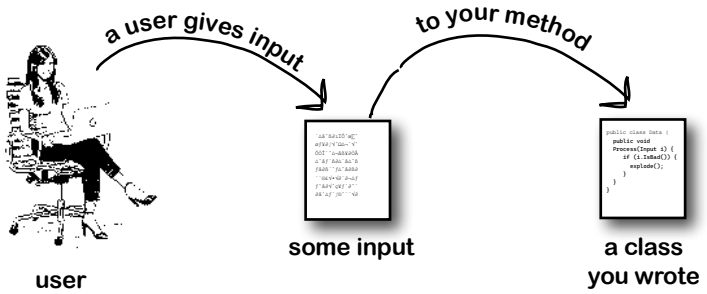
Actually, there *is* something you can do about it.

Yes, users do, indeed, screw up all the time. They'll give you bad data, weird input, and click on things you didn't even know existed. That's a fact of life, but that doesn't mean you can't do anything about it. C# gives you really useful **exception handling tools** to help you make your programs more robust. Because while you can't control what your users do with your app, you can make sure that your app doesn't crash when they do it.

What happens when a method you want to call is risky?

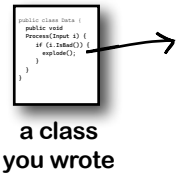
Users are unpredictable. They feed all sorts of weird data into your program, and click on things in ways you never expected. That's just fine because you can deal with exceptions that your code throws by adding **exception handling**, which lets you write special code that gets executed any time an exception is thrown.

1 Let's say a method called in your program takes input from a user.



2 That method could do something risky that might not work at runtime.

"Runtime" just means "while your program is running." Some people refer to exceptions as "runtime errors."

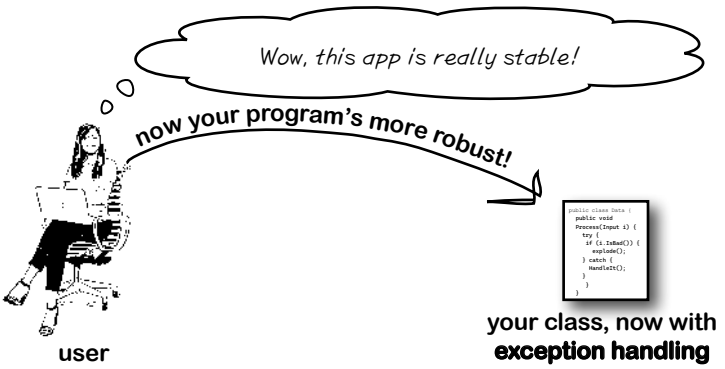



```
public void
   Process(Input i) {
   if (i.IsBad()) {
      Explode();
   }
}
```

3 You need to know that the method you're calling is risky.

If you can come up with a way to do a less risky thing that avoids throwing the exception, that's the best possible outcome! But some risks just can't be avoided, and that's when you want to do this.

4 You can then write code that can *handle* the exception if it does happen. You need to be prepared, just in case.



Handle exceptions with try and catch

When you add exception handling to your code, you're using the **try** and **catch** keywords to create a block of code that gets run if an exception is thrown.

Your **try/catch** code basically tells the C# compiler: "**Try** this code, and if an exception occurs, **catch** it with this *other* bit of code." The part of the code you're trying is the **try block**, and the part where you deal with exceptions is called the **catch block**. In the **catch** block, you can do things like print a friendly error message instead of letting your program come to a halt.

Let's have another look at the last three lines of the stack trace in our HexDump scenario to help us figure out where to put our exception handling code:

```
at System.IO.File.OpenRead(String path)
at HexDump.Program.GetInputStream(String[] args) in Program.cs:line 14
at HexDump.Program.Main(String[] args) in Program.cs:line 20
```

The UnauthorizedAccessException is caused by the line in GetInputStream that calls File.OpenRead. Since we can't prevent that exception, let's modify GetInputStream to use a **try/catch** block:

```
Stream GetInputStream(string[] args)
{
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
    {
        try
        {
            return File.OpenRead(args[0]);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.Error.WriteLine("Unable to read {0}, dumping from stdin: {1}",
                                    args[0], ex.Message);
            return Console.OpenStandardInput();
        }
    }
}
```

Put the code that might throw an exception inside the try block. If no exception happens, it'll get run exactly as usual, and the statements in the catch block will be ignored. If a statement in the try block throws an exception, the rest of the try block **won't get executed**.

This is the **try block**. You start exception handling with "try." In this case, we'll put the existing code in it.

The "catch" keyword signals that the block immediately following it contains an **exception handler**.

When an exception is thrown inside the try block, the program immediately jumps to the **catch** statement and starts executing the **catch block**.

We kept things simple in our exception handler. First we used Console.Error to write a line to the error output (stderr) letting the user know that an error occurred, then we fell back to reading data from standard input so the program still did something. Notice how **the catch block has a return statement**. The method returns a Stream, so if it handles an exception it still needs to return a Stream; otherwise, you'll get the "not all code paths return a value" compiler error.

Brain Power

If throwing an exception makes your code automatically jump to the catch block, what happens to the objects and data you were working with before the exception happened?

Use the debugger to follow the try/catch flow

An important part of exception handling is that when a statement in your **try** block throws an exception, the rest of the code in the block gets **short-circuited**. The program's execution immediately jumps to the first line in the **catch** block. Let's use the IDE's debugger to explore how this works.

Debug this!

1. Replace the GetInputStream method in your HexDump app with the one that we just showed you to handle an UnauthorizedAccessException.

2. Modify your project options to set the argument so that it contains the path to an unreadable file.

3. Place a breakpoint on the first statement in GetInputStream, then start debugging your project.

4. When it hits the breakpoint, step over the next few statements until you get to File.OpenRead. Step over it—the app jumps to the first line in the **catch** block.

Here's the breakpoint we placed on the first line of GetInputStream.

```
1 reference
Stream GetInputStream(string[] args)
{
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
    {
        try
        {
            return File.OpenRead(args[0]);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.Error.WriteLine("Unable to rea
                                    args[0], ex.Me
            return Console.OpenStandardInput();
        }
    }
}

using (Stream input = GetInputStream(args))
{
    var position = 0;
    var buffer = new byte[16];
    while (true)
    {
        // Read up to the next 16 bytes from the
        var bytesRead = input.Read(buffer, 0, buf
        if (bytesRead == 0) return;
```

Step through the method. When it hits File.OpenRead it will throw an exception, which will cause the execution to jump to the catch block.

Exception Caught

System.UnauthorizedAccessException: 'Access to the path 'C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\net8.0\binarydata.dat' is denied.'

This exception was originally thrown at this call stack:
[External Code]
Program.<Main>$.__GetInputStream|0_0(string[]) in Program.cs

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings
Break when this exception type is thrown
Except when thrown from:
HexDump.dll
Open Exception Settings | Edit Conditions

5. Keep stepping through the rest of the **catch** block. It will write the line to the console, and then return Console.OpenStandardInput() and resume the top-level statements.

If you have code that ALWAYS needs to run, use a finally block

When your program throws an exception, a couple of things can happen. If the exception ***isn't*** handled, your program will stop processing and crash. If the exception ***is*** handled, your code jumps to the `catch` block. What about the rest of the code in your `try` block? What if you were closing a stream, or cleaning up important resources? That code needs to run, even if an exception occurs, or you're going to make a mess of your program's state. That's where you'll use a **`finally` block**. It's a block of code that comes after the `try` and `catch`. The **`finally`** block always runs, whether or not an exception was thrown. Let's use the debugger to explore how the `finally` block works.

Debug this!

1 **Create a new Console App project called *ExceptionExperiment*.**
Add these top-level statements:

```
var firstLine = "No first line was read";
try
{
    var lines = File.ReadAllLines(args[0]);
    firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
}
catch (Exception ex)
{
    Console.Error.WriteLine("Could not read lines from the file: {0}", ex);
}
finally
{
    Console.WriteLine(firstLine);
}
```

WriteLine calls the Exception object's ToString method, which returns the exception name, message, and stack trace.

This finally block will run whether or not the try catches an exception.

You'll see the exception in the Locals window, just like you did earlier.

2 **Add a breakpoint to the first line of the top-level statements.**
Debug your app and step through it. The first line in the `try` block tries to access `args[0]`, but since you didn't specify any command-line arguments, the `args` array is empty and it throws an exception—specifically, a **System.IndexOutOfRangeException**, with the message *"Index was outside the bounds of the array."* After it prints the message, it **executes the `finally` block**, and then the program exits.

3 **Set a command-line argument with the path of a valid file.**
Use the project properties to pass a command-line argument to the app. Give it the full path of a valid file. Make sure there are no spaces in the filename; otherwise, the app will interpret it as two arguments. Debug your app again—after it finishes the `try` block, it **executes the `finally` block**.

4 **Set a command-line argument with an invalid file path.**
Go back to the project properties and change the command-line argument to pass the app the name of a file that does not exist. Run your app again. This time it catches a different exception: a **System.IO.FileNotFoundException**. Then it **executes the `finally` block**.

Catch-all exceptions handle System.Exception

You just made your console app throw two different kinds of exception—an IndexOutOfRangeException and a FileNotFoundException—and they were both handled. Take a closer look at the `catch` block:

```
catch (Exception ex)
```

This is a **catch-all exception**: the type after the `catch` block indicates what type of exception to handle, and since all exceptions extend the System.Exception class, specifying `Exception` as the type tells the `try/catch` block to catch any exception.

Avoid catch-all exception with multiple catch blocks

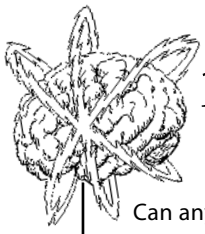
It's always better to try to anticipate the specific exceptions that your code will throw and handle them. For example, we know that this code can throw an IndexOutOfRange exception if no filename is specified, or a FileNotFound exception if an invalid file is found. We also saw earlier in the chapter that trying to read an unreadable file causes the CLR to throw an UnauthorizedAccessException. You can handle these different kinds of exceptions by adding **multiple `catch` blocks** to your code:

```
var firstLine = "No first line was read";
try
{
    var lines = File.ReadAllLines(args[0]);
    firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
}
catch (IndexOutOfRangeException)
{
    Console.Error.WriteLine("Please specify a filename.");
}
catch (FileNotFoundException)
{
    Console.Error.WriteLine("Unable to find file: {0}", args[0]);
}
catch (UnauthorizedAccessException ex)
{
    Console.Error.WriteLine("File {0} could not be accessed: {1}",
                                    args[0], ex.Message);
}
finally
{
    Console.WriteLine(firstLine);
}
```

This exception handler has three catch blocks to handle three different Exception types.

Only this catch block had to specify a variable name for the Exception object.

Now your app will write different error messages depending on which exception is handled. Notice that the first two `catch` blocks **did not specify a variable name** (like `ex`). You only need to specify a variable name if you're going to use the Exception object.



Brain Power

Can any of these exceptions be prevented rather than handled?

there are no Dumb Questions

Q: Back up a second. So every time my program runs into an exception, it's going to stop whatever it's doing unless I specifically write code to catch it? How is that a good thing?

A: One of the best things about exceptions is that they make it obvious when you run into problems. Imagine how easy it could be in a complex application for you to lose track of all of the objects your program was working with. Exceptions call attention to your problems and help you root out their causes so that you always know that your program is doing what it's supposed to do.

Any time an exception occurs in your program, something you expected to happen didn't. Maybe an object reference wasn't pointing where you thought it was, or a user entered a value you hadn't planned for, or a file you thought you'd be working with suddenly isn't available. If something like that happened and you didn't know it, it's likely that the output of your program would be wrong, and the behavior from that point on would be pretty different from what you expected when you wrote the program.

Now imagine that you had no idea the error had occurred and your users started calling you up and telling you that your program was unstable. That's why it's a good thing that exceptions disrupt everything your program is doing. They force you to deal with the problem while it's still easy to find and fix.

Q: Remind me again—what do I use the Exception object for?

A: The Exception object gives you clues about what went wrong. You can use its type to determine what kind of problem happened and write an exception handler to deal with it in a way that keeps your app running.

Q: What's the difference between a *handled* exception and an *unhandled* exception?

A: Whenever your program throws an exception, the runtime environment will search through your code looking for a `catch` block that handles it. If you've written one, the `catch` block will execute and do whatever you specified for that particular exception. Since you wrote a `catch` block to deal with that error up front, that exception is considered handled.

If the runtime can't find a `catch` block that matches the exception, it stops everything your program is doing and raises an error. That's an *unhandled* exception.

Q: What happens when you have a `catch` that doesn't specify a particular exception?

A: That's called a **catch-all exception**. A `catch` block like that will catch any kind of exception the `try` block can throw. So if you don't need to declare a variable to use the Exception object, an easy way to write a catch-all exception is like this:

```
catch
{
   // handle the exception
}
```

Q: Isn't it easier to use a catch-all exception? Isn't it safer to write code that always catches every exception?

A: You should **always do your best to avoid catching Exception,** and instead catch specific exceptions. You know that old saying about how an ounce of prevention is better than a pound of cure? That's especially true in exception handling. Depending on catch-all exceptions is usually just a way to make up for bad programming. For example, you're often better off using File.Exists to check for a file before you try to open it rather than catching a FileNotFoundException. While some exceptions are unavoidable, you'll find that a surprising number of them never have to be thrown in the first place.

Q: If a `catch` block with no specified exception will catch anything, why would I ever want to specify an exception type?

A: Certain exceptions might require different actions to keep your program moving. Maybe an exception caused by dividing by zero might have a `catch` block where it goes back and sets properties to save some of the data you've been working with, while a null reference exception in the same block of code might require it to create new instances of an object.

An unhandled exception can cause your program to run unpredictably. That's why the program stops whenever it runs into one.

Pool Puzzle

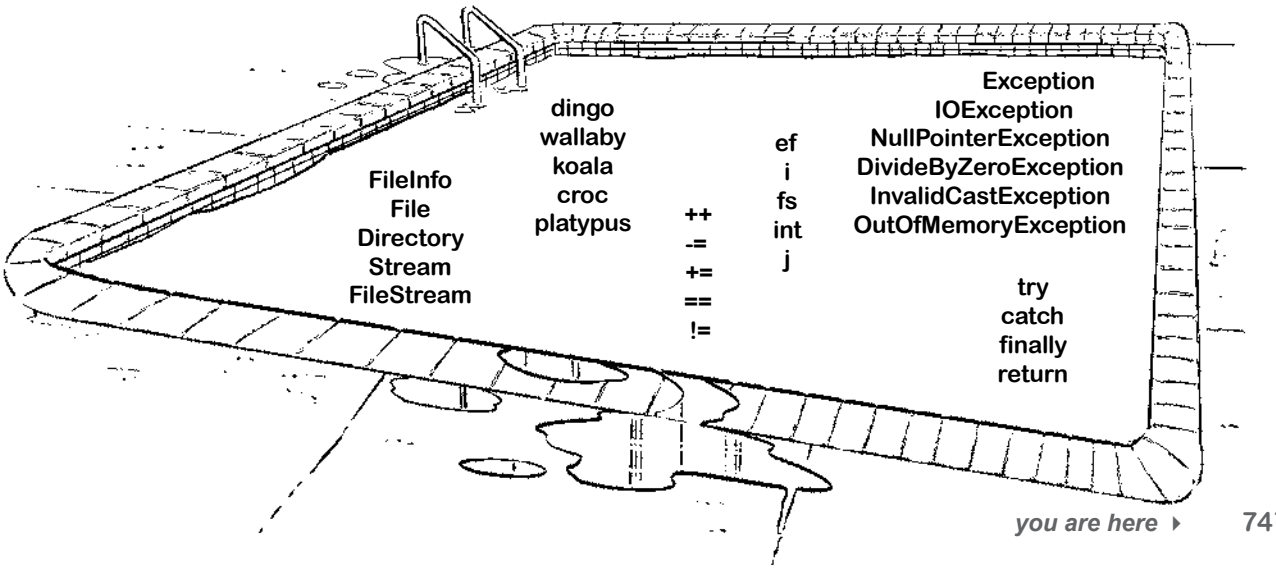
Your **job** is to take code snippets from the pool and place them into the blank lines in the program. You can use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the program produce the following output.

Output: → `G'day Mate!`

```
Kangaroo joey = new Kangaroo();
int koala = joey.Wombat(
           joey.Wombat(joey.Wombat(1)));
try {
    Console.WriteLine((15 / koala)
                  + " eggs per pound");
}
catch (______________________) {
        Console.WriteLine("G'Day Mate!");
}
```

```
class Kangaroo {
    ___________ fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        _______ ______;
        try {
            if (________ > 0) {
                fs = File.OpenWrite("wobbegong");
                croc = 0;
            } else if (________ < 0) {
                croc = 3;
            } else {
                _____ = ______.OpenRead("wobbegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (_______ > 2) {
                croc ______ dingo;
            }
        }
        ________ _______;
    }
}
```

Note: each snippet from the pool can be used more than once!



Pool Puzzle Solution

joey.Wombat is called three times, and the third time it returns zero. That causes WriteLine to throw a DivideByZeroException.

```
Kangaroo joey = new Kangaroo();
int koala = joey.Wombat(
            joey.Wombat(joey.Wombat(1)));
try {
    Console.WriteLine((15 / koala)
                    + " eggs per pound");
}
catch (__DivideByZeroException__) {
        Console.WriteLine("G'Day Mate!");
}

class Kangaroo {
    __FileStream__ fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        __dingo__ __++__;
        try {
            if (__wallaby__ > 0) {
                fs = File.OpenWrite("wobbegong");
                croc = 0;
            } else if (__wallaby__ < 0) {
                croc = 3;
            } else {
                __fs__ = __File__.OpenRead("wobbegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (__dingo__ > 2) {
                croc __-=__ dingo;
            }
        }
        __return__ __croc__;
    }
}
```

This catch block only catches exceptions where the code divides by zero.

The clue that this is a FileStream is that it has an OpenRead method and throws an IOException.

This code opens a file called "wobbegong" and keeps it open the first time it's called. Later on, it opens the file again. But it never closed the file, which causes it to throw an IOException.

You already know that you always have to close files when you're done with them. If you don't, the file will be locked open, and if you try to open it again, it'll throw an IOException.

Remember, you should avoid catch-all exceptions in your code. You should also avoid other things we do to make puzzles more interesting, like using obfuscated variable names.

Output: → **G'day Mate!**

*You keep talking about **risky code**, but isn't it risky to leave an exception unhandled? Why would I ever write an exception handler that doesn't handle **every type of exception**?*

Unhandled exceptions *bubble up*.

Believe it or not, it can be really useful to leave exceptions unhandled. Real-life programs have complex logic, and it's often difficult to recover correctly when something goes wrong, especially when a problem occurs very far down in the program. By only handling specific exceptions and avoiding catch-all exception handlers, you let unexpected exceptions **bubble up**: instead of being handled in the current method, they're caught by the next statement up the call stack. Anticipating and handling the exceptions that you expect and letting unhandled exceptions bubble up is a great way to build more robust apps.

Sometimes it's useful to **rethrow** an exception, which means that you handle an exception in a method but *still bubble it up* to the statement that called it. All you need to do to rethrow an exception is call `throw;` inside a `catch` block, and the exception that it caught will immediately bubble up:

```
try {
    // some code that might throw an exception
} catch (DivideByZeroException d) {
    Console.Error.WriteLine($"Got an error: {d.Message}");
    throw;
}
```

The throw command causes the DivideByZeroException to bubble up to whatever code called this try/catch block.

Here's a career tip: a lot of C# programming job interviews include a question about how you deal with exceptions in a constructor.

Watch it!

Keep risky code out of the constructor!

You've noticed by now that a constructor doesn't have a return value, not even `void`*. That's because a constructor doesn't actually return anything. Its only purpose is to initialize an object—which is a problem for exception handling inside the constructor. When an exception is thrown inside the constructor, then the statement that tried to instantiate the class **won't end up with an instance of the object**.*

Use the right exception for the situation

When you use the IDE to generate a method, it adds code that looks like this:

```
private void MyGeneratedMethod()
{
    throw new NotImplementedException();
}
```

The **NotImplementedException** is used any time you have an operation or method that hasn't been implemented. It's a great way to add a placeholder—as soon as you see one, you know there's code that you need to write. That's just one of the many exceptions that .NET provides.

Choosing the right exception can make your code easier to read, and make exception handling cleaner and more robust. For example, code in a method that validates its parameters can throw an ArgumentException, which has an overloaded constructor with a parameter to specify which argument caused the problem. Consider the Guy class back in Chapter 3. It had a ReceiveCash method that checked the `amount` parameter to make sure it was receiving a positive amount. This is a good opportunity to throw an ArgumentException:

```
public void ReceiveCash(int amount)
{
    if (amount <= 0)
       throw new ArgumentException($"Must receive a positive value", "amount");
    Cash += amount;
}
```

Pass the name of the invalid argument to the ArgumentException constructor.

Take a minute and look over the list of exceptions that are part of the .NET API—you can throw any of them in your code: *https://docs.microsoft.com/dotnet/api/system.systemexception*

Catch custom exceptions that extend System.Exception

Sometimes you want your program to throw an exception because of a special condition that could happen when it runs. Let's go back to the Guy class from Chapter 3. Suppose you're using it in an app that absolutely depended on a Guy always having a positive amount of cash. You could add a custom exception that **extends System.Exception**:

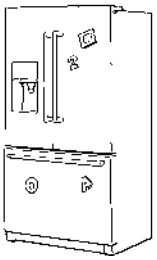
```
class OutOfCashException : System.Exception {
    public OutOfCashException(string message) : base(message) { }
}
```

This custom OutOfCashException extends the base System.Exception constructor with a message parameter.

Now you can throw that new exception, and catch it exactly like you'd catch any other exception:

```
class Guy
{
    public string Name;
    public int Cash;

    public int GiveCash(int amount)
    {
        if (Cash <= 0) throw new OutOfCashException($"{Name} ran out of cash");
        ...
```

Now that your Guy throws a custom exception, whatever method calls GiveCash can handle that exception in its own try/catch block.



Exception Magnets

Arrange the magnets so the application writes the following output to the console:

`when it thaws it throws.`

Here are the top-level statements for the app, and the MyException class used in the magnets:

```
Console.Write("when it ");
ExTestDrive.Zero("yes");
Console.Write(" it ");
ExTestDrive.Zero("no");
Console.WriteLine(".");

class MyException : Exception { }
```

```
}
```

```
throw new MyException();
```

```
}
```

```
      Console.Write("r");
   }
}
```

```
}
```

```
DoRisky(test);
```

```
} finally {
```

```
Console.Write("o");
```

```
Console.Write("s");
```

```
Console.Write("w");
```

```
class ExTestDrive {
   public static void Zero(string test)
{
```

```
static void DoRisky(String t) {
   Console.Write("h");
```

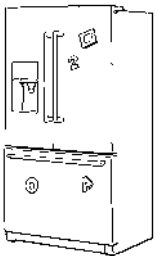
```
try {
```

```
if (t == "yes") {
```

```
Console.Write("a");
```

```
Console.Write("t");
```

```
} catch (MyException)
  {
```



Exception Magnets Solution

Arrange the magnets so the application writes the following output to the console:

when it thaws it throws.

Here are the top-level statements for the app, and the MyException class used in the magnets:

```
Console.Write("when it ");
ExTestDrive.Zero("yes");
Console.Write(" it ");
ExTestDrive.Zero("no");
Console.WriteLine(".");

class MyException : Exception { }

class ExTestDrive {
   public static void Zero(string test)
{
        try {
            Console.Write("t");
            DoRisky(test);
            Console.Write("o");
        } catch (MyException) {
            Console.Write("a");
        } finally {
            Console.Write("w");
        }
        Console.Write("s");
    }
    static void DoRisky(String t) {
       Console.Write("h");
        if (t == "yes") {
            throw new MyException();
        }
        Console.Write("r");
    }
}
```

We gave you this line to define a custom exception called MyException, which gets caught in a catch block in the code.

The Zero method either prints "thaws" or "throws," depending on whether it was passed "yes" or something else as its test parameter.

The finally block makes sure that the method always prints "w," and the "s" is printed outside the exception handler, so it always prints too.

The DoRisky method only throws an exception if it's passed the string "yes."

This line only gets executed if DoRisky doesn't throw the exception.

Behind the Scenes

IDisposable uses try/finally to ensure the Dispose method is called

Remember when we sleuthed out this code from Chapter 10?

```
using System.Text;

using (var ms = new MemoryStream())
{
    using (var sw = new StreamWriter(ms))
    {
        sw.WriteLine("The value is {0:0.00}", 123.45678);
    }
    Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
}
```

We moved this using statement inside the outer one to make sure the StreamWriter.Close method got called before MemoryStream.Close.

We were exploring how **using** statements work. In this case, we needed to nest one **using** statement inside the other to make sure the StreamWriter was disposed of before the MemoryStream. We did that because both the StreamWriter and MemoryStream classes implement the IDisposable interface and put a call to their Close method inside their Dispose method. Each **using** statement makes sure the Dispose method is called at the end of its block, and that ensures the streams are always closed.

Now that you've been working with exception handling, you can see how the **using** statement works. The **using** statement is an example of **syntactic sugar**: a way that C# "sweetens" the language by giving you a convenient shortcut that makes your code easier to read. When you do this, it's actually a kind of shortcut:

```
using (var sw = new StreamWriter(ms))
{
    sw.WriteLine("The value is {0:0.00}", 123.45678);
}
```

The C# compiler actually generates compiled code that looks (approximately) like this:

```
try {
    var sw = new StreamWriter(ms))
    sw.WriteLine("The value is {0:0.00}", 123.45678);
} finally {
    sw.Dispose();
}
```

Putting the Dispose statement in a **finally** block makes sure that it's always run—even if an exception occurs.

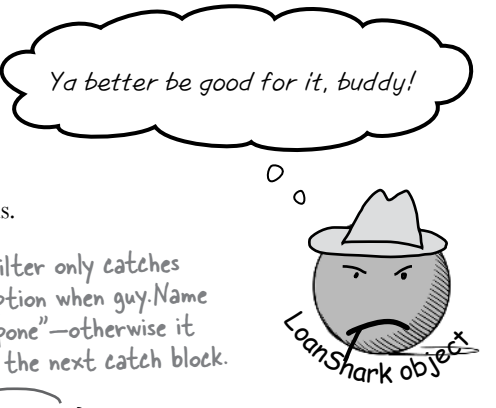
IDisposable is a great tool for avoiding exceptions.

Or anything else that implements IDisposable.

Avoid unnecessary exceptions...ALWAYS USE A USING BLOCK ANY TIME YOU USE A STREAM!

Exception filters help you create precise handlers

Let's say we're building a game set in classic 1930s mafia gangland, and we've got a LoanShark class that needs to collect cash from instances of Guy using the Guy.GiveCash method and that handles any OutOfCashException with a good old gangland-style lesson.

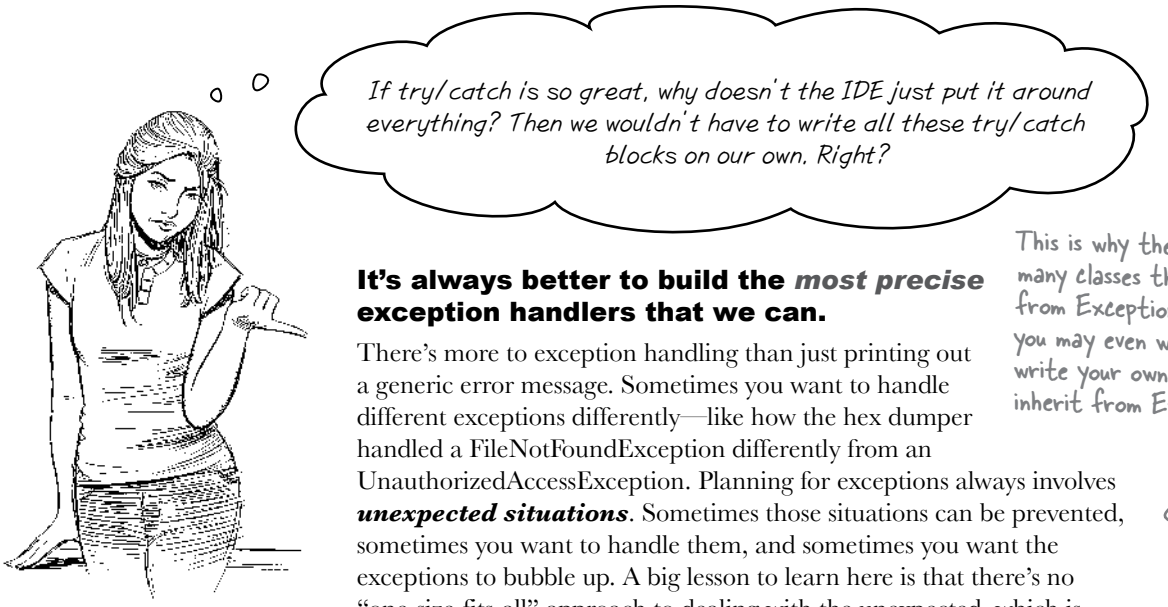


The thing is, every loan shark knows the golden rule: don't try to collect money from the big mob boss. That's where **exception filters** can come in really handy. An exception filter uses the `when` keyword to tell your exception handler to catch an exception only under specific conditions.

Here's an example of how an exception filter works:

This exception filter only catches OutOfCashException when guy.Name is set to "Al Capone"—otherwise it falls through to the next catch block.

```
try
{
    loanShark += guy.GiveCash(amount);
    emergencyReserves -= amount;
} catch (OutOfCashException) when (guy.Name == "Al Capone")
{
    Console.WriteLine("Don't mess with the mafia boss");
    loanShark += amount;
} catch (OutOfCashException ex)
{
    Console.Error.WriteLine($"Time to teach {guy.Name} a lesson: {ex.Message}");
}
```



It's always better to build the *most precise* exception handlers that we can.

There's more to exception handling than just printing out a generic error message. Sometimes you want to handle different exceptions differently—like how the hex dumper handled a FileNotFoundException differently from an UnauthorizedAccessException. Planning for exceptions always involves ***unexpected situations***. Sometimes those situations can be prevented, sometimes you want to handle them, and sometimes you want the exceptions to bubble up. A big lesson to learn here is that there's no "one-size-fits-all" approach to dealing with the unexpected, which is why the IDE doesn't just wrap everything in a `try/catch` block.

This is why there are so many classes that inherit from Exception, and why you may even want to write your own classes to inherit from Exception.

there are no Dumb Questions

Q: I'm still not clear on when I should be catching exceptions, when I should be preventing them, and when I should let them bubble up.

A: That's because there's no single correct answer or a perfect rule that you can follow in every situation—it always depends on what you want your code to do.

Actually, that's not 100% true. There is a rule: it's always better to prevent exceptions if you can. You can't always anticipate the unexpected, especially when you're dealing with input from users or decisions they make.

Deciding whether to let exceptions bubble up or handle them in the class often boils down to separation of concerns. Does it make sense for a class to know about a certain exception? It depends on what that class does. Luckily, the IDE's refactoring tools are always there to help you change your code if you decide that certain exceptions are better off bubbled up than caught.

Q: Can you explain what you meant by "syntactic sugar"?

A: When developers use the term "syntactic sugar," they're typically talking about a programming language giving you a convenient and easy-to-understand shortcut for code that would otherwise be more complex. The word "syntax" refers to the keywords of C# and the rules that govern them. The `using` statement is an official part of the C# syntax, which has rules that say that it must be followed by a variable declaration that instantiates a type that implements IDisposable, followed by a block of code. The "sugar" part refers to the fact that it's *super sweet* that the C# compiler turns it into something that would be much clunkier if you had to write it out by hand.

Q: Is it possible to use an object with a `using` statement if it doesn't implement IDisposable?

A: No. You can only create objects that implement IDisposable with `using` statements, because they're tailor-made for each other. Adding a `using` statement is just like creating a new instance of a class, except that it always calls its Dispose method at the end of the block. That's why the class **must implement** the IDisposable interface.

Q: Can you put any statement inside a `using` block?

A: Definitely. The whole idea with `using` is that it helps you make sure that every object you create with it is disposed of. But what you do with those objects is entirely up to you. In fact, you can create an object with a `using` statement and never even use it inside the block. That would be pretty useless, though, so we don't recommend doing that.

Q: Can you call the Dispose method outside of a `using` statement?

A: Yes. You don't ever actually need to use a `using` statement. You can call Dispose yourself when you're done with the object, or manually do whatever cleanup is necessary—like calling a stream's Close method, which you did in Chapter 10. If you use a `using` statement, it'll make your code easier to understand and prevent problems that happen if you don't dispose of your objects.

Q: Since a `using` statement basically generates a `try/catch` block that calls the Dispose method, is it OK to do exception handling inside of the `using` block?

A: Yes. It works exactly like the nested `try/catch` blocks that you used earlier in the chapter in your GetInputStream method.

Q: You mentioned a `try/finally` block. Does that mean it's OK to have a `try` and `finally` without a `catch`?

A: Yes! You can definitely have a `try` block without a `catch`, and just a `finally`. It looks like this:

```
try {
     DoSomethingRisky();
     SomethingElseRisky();
}
finally {
     AlwaysExecuteThis();
}
```

If DoSomethingRisky throws an exception, then the `finally` block will immediately run.

Q: Does Dispose only work with files and streams?

A: Not at all. There are a lot of classes that implement IDisposable, and when you're using one you should always use a `using` statement. If you write a class that has to be disposed of in a certain way, then you can implement IDisposable too.

There's no one-size-fits-all approach to planning for the unexpected.

The worst catch block EVER: catch-all plus comments

A `catch` block will let your program keep running if you want it to. An exception gets thrown, you catch the exception, and instead of shutting down and giving an error message, you keep going. But sometimes, that's not such a good thing.

Take a look at this Calculator class, which seems to be acting funny all the time. What's going on?

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch {

            // TODO: we need to figure out a way to prevent our
            // users from entering a zero in a division problem.

        }
    }
}
```

Here's the problem. If divisor is zero, this will create a DivideByZeroException.

The programmer thought that they could bury the exceptions by using an empty catch block, but they just caused a headache for whoever had to track down problems with it later.

You should handle your exceptions, not bury them

Just because you can keep your program running doesn't mean you've *handled* your exceptions. In the previous code, the calculator won't crash…at least, not in the Divide method. What if some other code calls that method, and tries to print the results? If the divisor was zero, then the method probably returned an incorrect (and unexpected) value.

Instead of just adding a comment and burying the exception, you need to **handle the exception**. If you're not able to handle the problem, ***don't leave empty or commented*** `catch` ***blocks!*** That just makes it harder for someone else to track down what's going on. It's better to let the program continue to throw exceptions, because then it's easier to figure out what's going wrong.

Remember, when your code doesn't handle an exception, the exception bubbles up the call stack. Letting an exception bubble up is a perfectly valid way of dealing with an exception, and in some cases it makes more sense to do that than to use an empty catch block to bury the exception.

Temporary solutions are OK (temporarily)

Sometimes you find a problem, and know it's a problem, but aren't sure what to do about it. In these cases, you might want to log the problem and note what's going on. That's not as good as handling the exception, but it's better than doing nothing.

Here's a temporary solution to the calculator problem:

...but in real life, "temporary" solutions have a nasty habit of becoming permanent.

Take a minute and think about this `catch` block. What happens if the `StreamWriter` can't write to the C:\Logs\ folder? You can nest another `try/catch` block inside it to make it less risky. Can you think of a better way to do this?

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch (Exception ex) {
            using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.
txt");
            {
                sw.WriteLine(ex.getMessage());
            }

        }
    }
}
```

This still needs to be fixed, but short-term, this makes it clear where the problem occurred. Still, wouldn't it be better to figure out why your Divide method is being called with a zero divisor in the first place?

I get it. It's sort of like using exception handling to place a marker in the problem area.

Handling exceptions doesn't always mean the same thing as FIXING exceptions.

It's never good to have your program bomb out. It's way worse to have no idea why it's crashing or what it's doing to users' data. That's why you need to be sure that you're always dealing with the errors you can predict and logging the ones you can't. While logs can be useful for tracking down problems, preventing those problems in the first place is a better, more permanent solution.

Is it really practical to write a new file every time my code catches an exception? If my app is big, it could generate a lot of information from those exception handlers. There's got to be a ***better way to manage all that information****... right?*

Logging helps you track, debug, and improve your apps.

Take a minute and think back over everything you've done on your C# journey so far. You've written a lot of code! You've created serious business apps, Unity games, .NET MAUI applications...the folder where you saved all of your projects must be getting pretty big by now.

One thing you *haven't* done in this book is give your code to users. All of the projects in this book are there to help you learn, practice, and explore C#, .NET, and important development principles, ideas, and concepts. When you finish a project, you can save it in version control and move on to the next one.

That's where **logging** comes in. Logging means adding code that creates messages any time important events happen. When an exception or other error happens, your app can log it. You can also log important actions and events, like when a user takes an action or a class accesses a specific resource. Logging is also really useful for ***tracking down bugs***. Developers will often add debug logging, adding lots of extra log messages that can be turned on to generate a lot of extra information. If your app crashes or has a bug, you can look at the logs to see what went wrong.

Use NuGet to add packages to your project

In the last part of this chapter, you'll add a logging library to a .NET project. We'll be using the terms "library" and "package"—they're similar, but they mean different things:

- A **library** is a collection of classes, enums, records, and other C# types and code. It lives in your project as a DLL—a binary file with the *.dll extension—that contains the compiled code.
- A **package** is a collection of one or more DLL files grouped together and given a name.

You can add packages to your project by **installing them with NuGet**, the **.NET package manager**. NuGet has a huge collection of over 100,000 packages used by millions of developers every day. You'll use Visual Studio or the "dotnet add package" command to fetch a logging library package from NuGet, which automatically downloads it over the internet and adds it to your project.

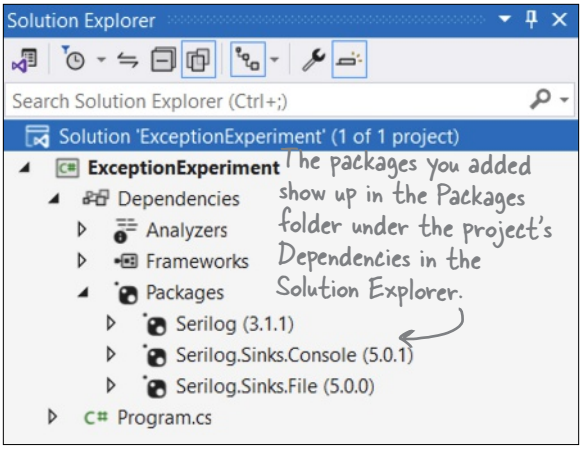
Use NuGet to add a logging library to your app

A **logging library** is a tool to help you write and manage logs. At the time we're writing this, there are three popular logging libraries that C# developers use: **Serilog**, **log4net**, and **NLog**. All three of those logging libraries are ***free and open source***, which means that you can use them in your apps, and even modify the code for them if you want. All three of those libraries are great—they all have similar features, but work slightly differently. We need to choose one to get started, so we'll **add Serilog to your ExceptionExperiment app** from earlier in the chapter.

Open your ExceptionExperiment app. Choose View >> Terminal (Ctrl-`) from the menu and enter these commands:

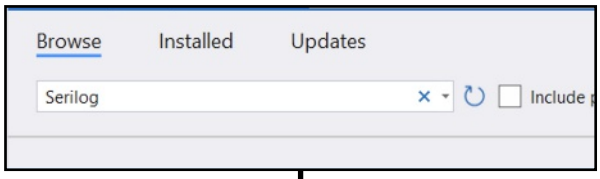
```
cd ExceptionExperiment
dotnet add package Serilog
dotnet add package Serilog.Sinks.Console
dotnet add package Serilog.Sinks.File
```

After each dotnet command, you should see a lot of messages, many of which will include URLs at nuget.org as the package manager figures out what files it needs to download. After both "add package" commands finish, go to the Solution Explorer and expand the Dependencies folder and the Packages folder underneath it. You should see three Serilog dependencies in your project (the versions might not match the ones in our screenshots).

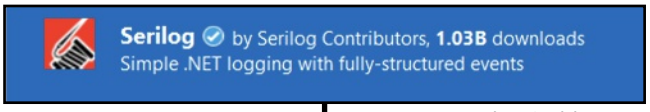


Visual Studio also has a NuGet package manager. Right-click on Dependencies in the Solution Explorer and choose Manage NuGet Packages... from the menu. Click Browse, search for Serilog, then install the Serilog, Serilog.Sinks.Console, and Serilog.Sinks.File packages. After they're installed, you should see them in the Packages folder in the Solution Explorer.

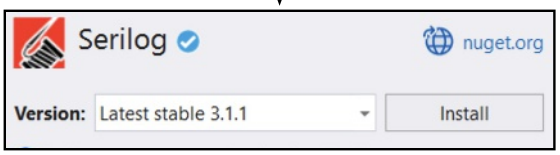
Search for packages in the NuGet package manager.



When you find a package, click on it to see details.



Click the Install button, then accept any licenses.



Packages add DLLs with classes

In Chapter 9, you added a second project to your solution with unit tests, and added a project dependency so it could use classes in the first project. Packages work the same way—each package contains classes that your project can access. The classes are added to your project as **DLL (dynamic link library) files**, or files that contain compiled C# code.

You can see the DLL files by building your app and then opening the bin\Debug\net8.0 folder—you'll see your ExceptionExperiment executable, a DLL with the same name, and three DLLs that correspond to the packages.

Add logging to your ExceptionExperiment app

When you add a package to a project, it adds classes in a namespace to your project—in this case, the Serilog namespace—so you'll start by adding a `using Serilog;` directive. It's convenient to see log messages written to the console as your app runs, but it's also very useful to have them written to files. You'll add lines at the start of the app to configure your logger to tell it to write all of its log messages to the console, and also to create a new file called *log-current-date-and-time.log* and write all messages to that file too.

Here's the code for your updated ExceptionExperiment app:

```
using Serilog;

Log.Logger = new LoggerConfiguration()
    .WriteTo.Console()
    .WriteTo.File($"log-{DateTime.Now:yyyyMMdd-HHmmss}.log")
    .CreateLogger();

Log.Information("App started with {0} argument(s): {1}", args.Length, args);

var firstLine = "No first line was read";
try
{
    var lines = File.ReadAllLines(args[0]);
    firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
    if (lines.Length == 0) Log.Warning("No lines found in file");
}
catch (IndexOutOfRangeException)
{
    Log.Warning("Please specify a filename.");
}
catch (FileNotFoundException)
{
    Log.Error("Unable to find file: {0}", args[0]);
}
catch (UnauthorizedAccessException ex)
{
    Log.Error(ex, "File {0} could not be accessed", args[0]);
}
finally
{
    Console.WriteLine(firstLine);
}

Log.CloseAndFlush();
```

Serilog uses a static class called Log. You'll configure it with chained methods. We're telling it to write all of its log messages to the console, and also to a new file with a filename that has the date and time the app was started.

We're using three static methods in the Serilog Log object that correspond to three *log levels*, or categories that indicate how important or urgent a message is. Information has the lowest level, providing general information about what the app is doing. Warning is the middle level, telling us about things that could potentially be problems. Error is the highest level, and is used to report something that's gone wrong—like a critical exception that we need to know about.

The Information, Warning, and Error methods work just like Console.WriteLine—you can replace the Console.WriteLine messages with them. The methods have an overload that takes an Exception object as the first parameter, which can be useful when logging exceptions because it can tell you the specific line of code that threw the exception.

When the app is done, it calls CloseAndFlush to write any remaining log messages and then close out the log.

Bullet Points

- Any statement can throw an **exception** if something fails at **runtime** (or while the code is running).
- Use a **`try/catch`** block to handle exceptions. Unhandled exceptions will cause your program to stop execution and pop up an error window.
- Any exception in the block of code after the `try` statement will cause the program's execution to immediately jump to the first statement in the **exception handler**, or the block of code after `catch`.
- The **Exception object** gives you information about the exception that was caught. If you specify an Exception variable in your `catch` statement, that variable will contain information about any exception thrown in the `try` block:

```
try {
  // statements that might
  // throw exceptions
} catch (IOException ex) {
  // if an exception is thrown,
  // ex has information about it
}
```

- There are many **different kinds of exceptions** that you can catch. Each has its own object that extends System.Exception.
- Try to avoid **catch-all exception handlers** just catching Exception. Handle specific exceptions instead.
- A **`finally` block** after the exception handlers will always run whether or not an exception is thrown.
- Each `try` can have more than one `catch`:

```
try { ... }
catch (NullReferenceException ex) {
   // these statements will run if a
   // NullReferenceException is thrown
}
catch (OverflowException ex) { ... }
catch (FileNotFoundException) { ... }
catch (ArgumentException) { ... }
```

- Your code can **throw an exception** using `throw`:

```
throw new Exception("Message");
```

- Your code can also **rethrow** an exception using `throw;` but this only works inside of a `catch` block. Rethrowing an exception preserves the call stack.
- You can create a **custom exception** by adding a class that extends the System.Exception base class:

```
class CustomException : Exception { }
```

- Most of the time, you only need to throw exceptions that are built into .NET, like `ArgumentException`. One reason to use different kinds of exceptions is to **give more information** to people troubleshooting problems.
- An **exception filter** uses the `when` keyword to tell your exception handler to catch an exception only under specific conditions.
- The `using` statement is **syntactic sugar** that causes the C# compiler to generate the equivalent of a `finally` block that calls the Dispose method.
- A **package** adds compiled code to your project that can include classes, types, and other resources for you to use in your own code.
- Packages are typically distributed with **NuGet**, the package manager for C# and .NET that automatically downloads packages and adds them to your project.
- **Logging** lets you record events, activities, errors, and other messages in your app. You can add a logging package like Serilog, NLog, or log4net to your project.

Use AI chatbots to research the *top five things* we wanted to teach you (but didn't have room for)

You're almost at the end of the book! You've learned so many things, and written a lot of code. You've also learned a lot about how to keep learning. That's really important, because while this book includes the *really important C# concepts, tools, and techniques* that you need to know to be a great developer, there just aren't enough pages for us to include every single thing that C# does. Here are the top five things we wanted to include in this book, but we ***just didn't have enough pages*** to cover them properly. This is your chance to practice prompt engineering skills and jump-start your next C# research and learning. We've provided some prompts to help get you started.

Packaging your apps so they can be installed—and doing everything from the command line

We've talked a lot about creating, debugging, and testing C# apps. But how do you actually get them to the people who need to use them? You need a way to get your apps packaged and installed on your users' computers.

- How do I create a self-contained .NET application for Windows, macOS, or Linux?
- How do I package and distribute my .NET application so users can install it?
- How do I create a new project from the command line? How do I run, test, and package it?

Tuples

You learned about anonymous types with LINQ. C# has a similar feature, a tuple—it's a way to create a new type by grouping together a fixed number of existing types. It can be a useful way to pass multiple kinds of data at once.

- What is a tuple in C#? When would I use a tuple in my C# app?
- How is a tuple different from an anonymous type? When would I use a tuple instead of an anonymous type?

Advanced pattern matching with switch expressions

We spent some time talking about switch expressions. But you can unlock really powerful ways to analyze a value and act on many different conditions in a single switch expression.

- What is pattern matching in C# switch expressions? How do I provide multiple inputs to a switch expression?
- Show me an example of a list pattern in C# to check elements in a list. How do I use relational patterns?

Lambda block bodies

We gave you a solid starting point with lambda expressions. But did you know that lambda expressions can have block bodies, which let you include multiple lines of code in a lambda expression? That can be a game-changer.

- Show me an example of a lambda block body in C# that executes multiple lines of code and returns a value.
- How do I use Visual Studio's Quick Fix feature to convert a lambda expression to a block body?

Discards

Sometimes you need to write an expression that needs a variable name, but you don't actually need to use the value in the variable, and that's where a discard comes in. It sounds simple, but it's actually a really useful tool.

- Show me an example of discards in C#. How can I use discards with lambda expressions?
- Give me examples of situations in C# where I need to provide a variable but don't need to use it.

Unity Lab #6

Scene Navigation

In the last Unity Lab, you created a scene with a floor (a plane) and a player (a sphere nested under a cylinder), and you used a NavMesh, a NavMesh Agent, and raycasting to get your player to follow your mouse clicks around the scene.

Now we'll pick up where the last Unity Lab left off. The goal of these labs is to get you familiar with Unity's **pathfinding and navigation system**, a sophisticated AI system that lets you create characters that can find their way around the worlds you create. In this lab, you'll use Unity's navigation system to make your GameObjects move themselves around a scene.

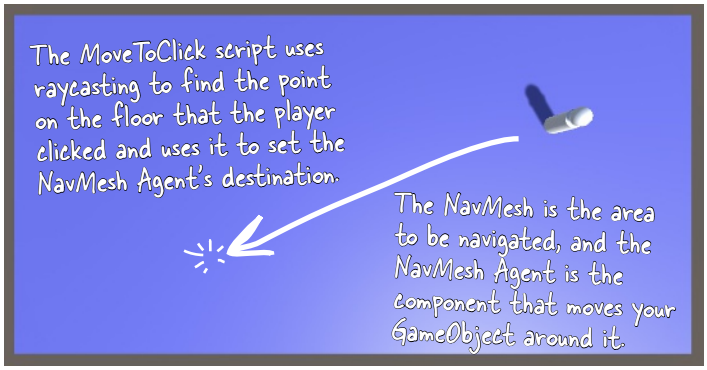
Along the way you'll learn about useful tools: you'll create a more complex scene and bake a NavMesh that lets an agent navigate it, you'll create static and moving obstacles, and most importantly, you'll **get more practice writing C# code in Unity**.

Let's pick up where the last Unity Lab left off

In the last Unity Lab, you created a player out of a sphere head nested under a cylinder body. Then you added a NavMesh Agent component to move the player around the scene, using raycasting to find the point on the floor that the player clicked. In this lab, you'll pick up where the last one left off. You'll add GameObjects to the scene, including stairs and obstacles so you can see how Unity's navigation AI handles them. Then you'll add a moving obstacle to really put that NavMesh Agent through its paces.

So go ahead and **open the Unity project** that you saved at the end of the last Unity Lab. If you've been saving up the Unity Labs to do them back-to-back, then you're probably ready to jump right in! But if not, take a few minutes and flip through the last Unity Lab again—and also look through the code that you wrote for it.

If you're using our book because you're preparing to be a professional developer, being able to go back and read and refactor the code in your old projects is a really important skill—and not just for game development!



there are no Dumb Questions

Q: There were a lot of moving parts in the last Unity Lab. Can you go through them again, just so I'm sure I have everything?

A: Definitely. The Unity scene you created in the last lab has four separate pieces. It's easy to lose track of how they work together, so let's go through them one by one:

1. First there's the **NavMesh**, which defines all of the "walkable" places your player can move around on in the scene. You made this by setting the floor as a walkable surface and then "baking" the mesh.

2. Next there's the **NavMesh Agent**, a component that can "take over" your GameObject and move it around the NavMesh just by calling its SetDestination method. You added this to your Player GameObject.

3. The camera's **ScreenPointToRay method** creates a ray that goes through a point on the screen. You added code to the Update method that checks if the player is currently pressing the mouse button. If so, it uses the current mouse position to compute the ray.

4. **Raycasting** is a tool that lets you cast (basically "shoot") a ray. Unity has a useful Physics.Raycast method that takes a ray, casts it up to a certain distance, and if it hits something tells you what it hit.

Q: So how do those parts work together?

A: Whenever you're trying to figure out how the different parts of a system work together, ***understanding the overall goal*** is a great place to start. In this case, the goal is to let the player click anywhere on the floor and have a GameObject move there automatically. Let's break that down into a set of individual steps. The code needs to:

- **Detect that the player clicked the mouse.**
Your code uses Input.GetMouseButtonDown to detect a mouse click.

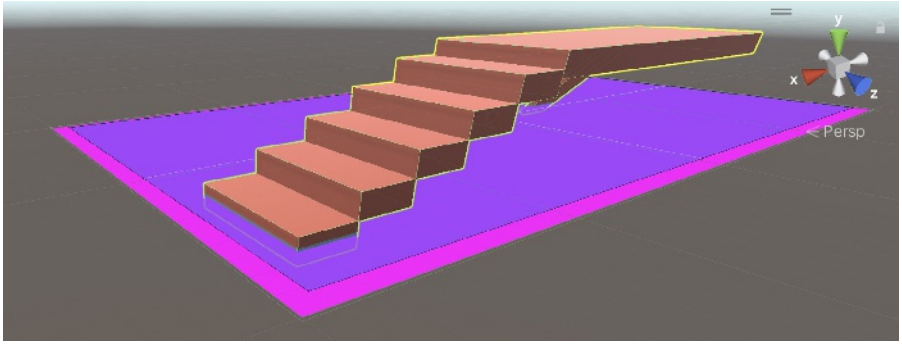
- **Figure out what point in the scene that click corresponds to.**
It uses Camera.ScreenPointToRay and Physics.Raycast to do raycasting and figure out which point in the scene the player clicked.

- **Tell the NavMesh Agent to set that point as a destination.**
The NavMeshAgent.SetDestination method triggers the agent to calculate a new path and start moving toward the new destination.

The MoveToClick method was adapted from *code on the Unity Manual page* for the NavMeshAgent.SetDestination method. Take a minute and read it right now—choose *Help >> Scripting Reference* from the main menu, then search for NavMeshAgent.SetDestination.

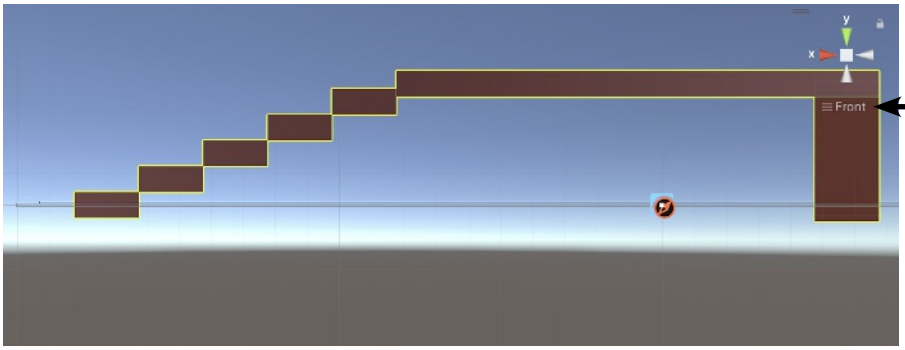
Add a platform to your scene

Let's do a little experimentation with Unity's navigation system. To help us do that, we'll add more GameObjects to build a platform with stairs, a ramp, and an obstacle. Here's what it will look like:



Sometimes it's easier to see what's going on in your scene if you switch to an isometric view. You can always reset the layout if you lose track of the view.

It's a little easier to see what's going on if we switch to an **isometric** view, or a view that doesn't show perspective. In a **perspective** view, objects that are farther away look small, while closer objects look large. In an isometric view, objects are always the same size no matter how far away they are from the camera.

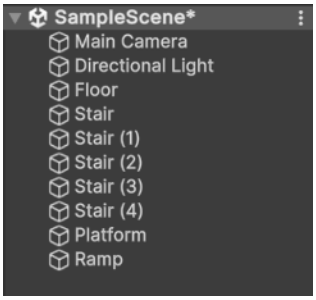


When you start Unity, the label (Persp) under the Scene Gizmo shows the view name. The three lines ( ) indicate that the Gizmo is in *perspective* mode. Click the cones to change the view to "Front" (Front). If you click the lines, they'll change to three parallel lines ( ), which toggles the view to *isometric* mode.

Add 10 GameObjects to your scene. **Create a new material called *Platform Material*** in your Materials folder with albedo color CC472F, and add it to all of the GameObjects. This table shows their names, types, and positions:

Name	Type	Position	Rotation	Scale
Stair	Cube	(17.25, 0, 5)	(0, 0, 0)	(2, 0.8, 5)
Stair (1)	Cube	(15.25, 0.8, 5)	(0, 0, 0)	(2, 0.8, 5)
Stair (2)	Cube	(13.25, 1.6, 5)	(0, 0, 0)	(2, 0.8, 5)
Stair (3)	Cube	(11.25, 2.4, 5)	(0, 0, 0)	(2, 0.8, 5)
Stair (4)	Cube	(9.25, 3.2, 5)	(0, 0, 0)	(2, 0.8, 5)
Platform	Cube	(0.75, 3.75, 5)	(0, 0, 0)	(15, 0.8, 5)
Ramp	Cube	(–5.75, 1.75, 0.75)	(–46, 0, 0)	(2, 0.25, 6)

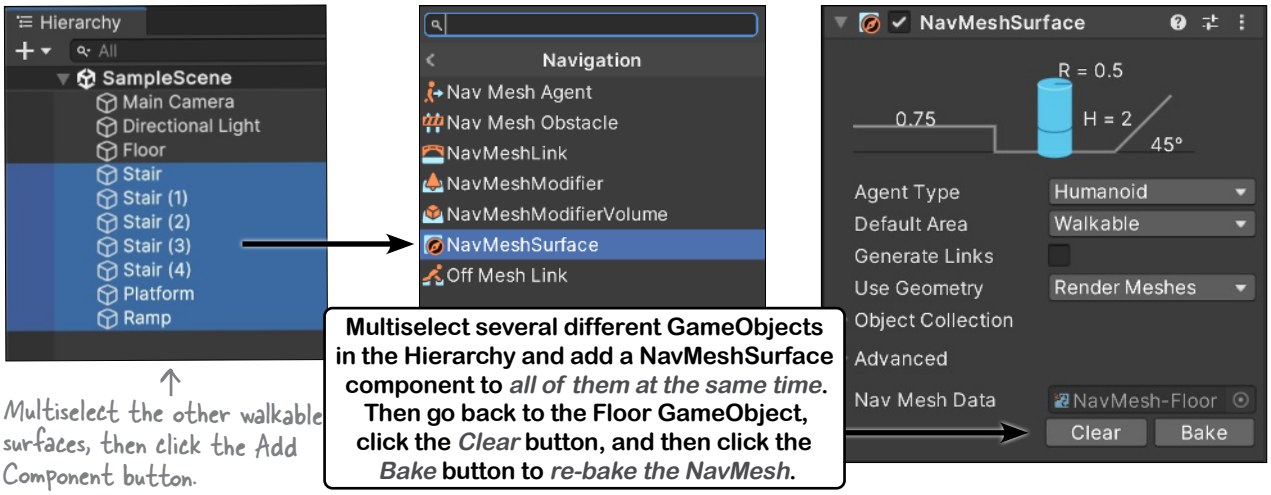
Add the first stair, then copy and paste it four times (or choose Duplicate from the menu).



Use bake options to make the platform walkable

Use Shift-click to **select all of the new GameObjects** that you added to the scene. Go to the Navigation window and click the Object button, then **make them all walkable**. Go to the Inspector window, click the Add Component, and choose Navigation >> NavMeshSurface to **add a NavMeshSurface component to all of them at once**.

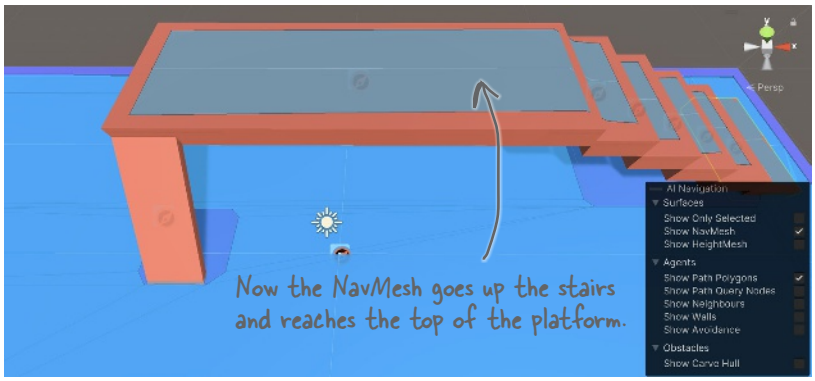
The scene still has the old NavMesh. Select the Floor GameObjects, **click the Clear button** to clear the NavMesh, then **click the Bake button to *re-bake* NavMesh.** This will create a new NavMesh for the scene.

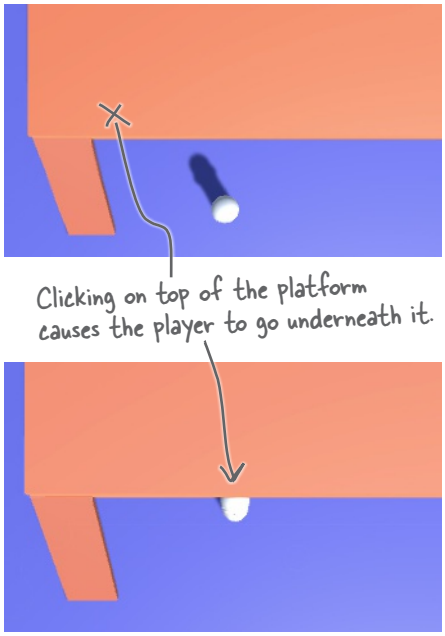


Multiselect the other walkable surfaces, then click the Add Component button.

Multiselect several different GameObjects in the Hierarchy and add a NavMeshSurface component to *all of them at the same time*. Then go back to the Floor GameObject, click the *Clear* button, and then click the *Bake* button to *re-bake the NavMesh*.

It looks like it worked! The NavMesh now appears on the platform and stairs. Run the game, click the top of the platform, and see what happens. Your GameObject should be able to walk up the stairs to the top of the platform.



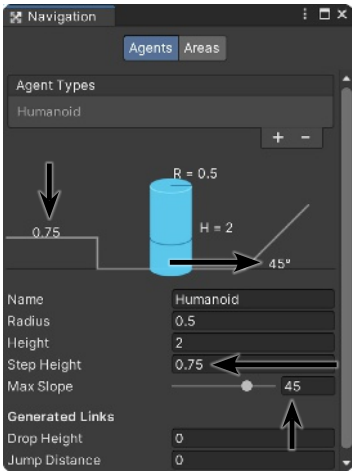


Hmm, hold on. Things aren't working the way we expected them to. When you click the top of the platform, the player goes ***under*** it.

If you look closely at the NavMesh on the stairs, you'll see a rectangle of NavMesh on each of them, but it's not connected from stair to stair and there's space around the NavMesh, so the agent doesn't have a continuous path up to the platform. And if you look at the ramp, you'll see that it doesn't have any NavMesh on it at all.

Include the stairs and ramp in the NavMesh

An AI that couldn't get your player up or down a ramp or stairs wouldn't be very intelligent. Luckily, Unity's pathfinding system can handle both—we just need to make a few small adjustments to the options when we bake the NavMesh.



Open the **Navigation panel** by choosing Window >> AI >> Navigation from the menu. Find the Step Height value—its default value is 0.75, and it appears on the left side of the picture and in the input boxes below it. The steps in your scene are 0.8 units tall, so **change Step Height to 0.8** so your player can navigate up them.

Next, find the Max Slope value, which defaults to 45 degrees. The X rotation value of your ramp is set to -46, which means it has a 46-degree incline. **Change the Max Slope value to 46** so your player can go up the ramp.

Now clear all of the NavMesh data from the scene:

1. Click on Floor in the Hierarchy panel and shift-click on Ramp to **multi-select all of the GameObjects that have a NavMesh**. Go to the NavMeshSurface component in the Inspector panel and click Clear.
2. De-select the GameObjects, then **select just the Floor GameObject**. Click the Bake button on just that one GameObject's NavMeshSurface component to **bake a new NavMesh for the whole scene**.

The NavMesh should now go up the stairs and the ramp. Start your game and test out your new NavMesh changes. Click on the platform, stairs, or ramp, and the player navigates to them. Change the camera to see under the platform—click there, and the player navigates underneath it. Your navigation works now!

Exercise

Add two obstacles to your scene: a sphere under the platform and a block that moves up and down when clicked.

- **Add a sphere named *8 Ball*** to your scene with position (5, 1.5, -5) and scale (3, 3, 3). Add a material to your scene called *8 Ball Material* that uses the 8 Ball Texture image as its albedo map and drag it onto the sphere.
- **Add a cube to your scene named *Moving Obstacle*** with position (–5.75, 1, –1) and scale (2, 2, 0.25). Create a new material for it named *Moving Obstacle Material* with a dark gray color (333333).

Add a new script to *Moving Obstacle* called MoveObstacle. The script uses the OnMouseDrag method. It works just like the OnMouseDown method you used in the last lab, except that it's called when the GameObject is dragged.

- Drag the script onto *Moving Obstacle*. Use this code to make it move when the mouse drags it up and down:
```
transform.position += new Vector3(0, Input.GetAxis("Mouse Y"), 0);
```
- Can you figure out how to use its transform.position.y property to keep from moving above 5 or below 1?

Modify the MoveToClick script to exclude obstacles. Set the tags for *8 Ball* and *Moving Obstacle* like in the last lab, but this time choose "Add tag" from the dropdown, then use the + button to add a new tag called Obstacle. Use the dropdown to assign the new tag to each GameObject. The Physics.Raycast method has an `out` parameter called `hit` with information about the GameObject the cast ray hit. Use `hit.collider.gameObject.tag` to find its tag—if it's set to `Obstacle`, don't navigate to it.

It's not cheating to peek at the solution!

Exercise Solution

Your MoveCamera script uses Input.GetAxis to respond to the scroll wheel. Now you're using the mouse's up-down movement—along the Y axis—to move the obstacle by modifying its Y position.

Here's the OnMouseDrag method for the MoveObstacle script.

```
void OnMouseDrag()
{
    transform.position += new Vector3(0, Input.GetAxis("Mouse Y"), 0);
    if (transform.position.y < 1)
    {
        transform.position = new Vector3(transform.position.x, 1, transform.position.z);
    }
    if (transform.position.y > 5)
    {
        transform.position = new Vector3(transform.position.x, 5, transform.position.z);
    }
}
```

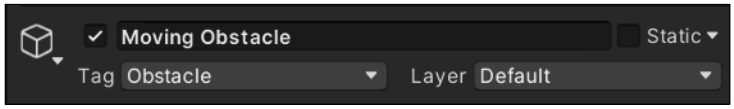
The first `if` statement keeps the block from moving below the floor, and the second keeps it from moving too high.

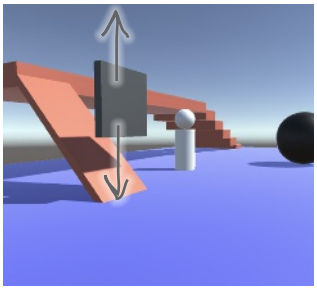
Here are the **changes to your MoveToClick script** that prevent it from navigating when an obstacle is clicked:

```
if (Physics.Raycast(ray, out hit, 100))
{
    if (hit.collider.gameObject.tag != "Obstacle")
    {
        agent.SetDestination(hit.point);
    }
```

We learned about out parameters in Chapter 10. The Raycast method uses the `out hit` parameter to return a struct called RaycastHit. Its hit.collider property contains a reference to the GameObject that was hit, and its tag property returns the tag.

When you run your game, click on the *Moving Obstacle* cube and *8 Ball* sphere. The player shouldn't move because of the `if` statement you added to MoveToClick. If it does move, double-check that you have their tags set to Obstacle:





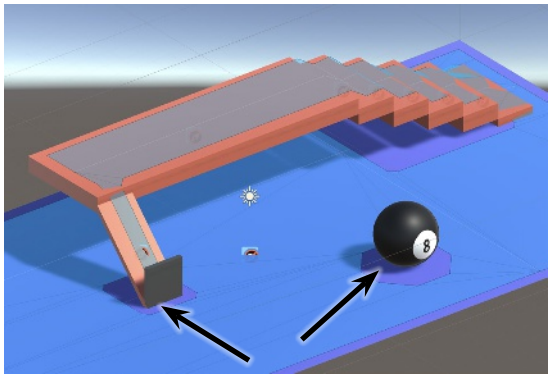
Now click and drag *Moving Obstacle* up and down. The obstacle should move up until it reaches a height of 5, and down until it hits the floor. The `if` statements in the MoveObstacle script work like the one in the MoveCamera script that keeps the camera above the floor, setting transform.position if the y value gets too big or small.

there are no Dumb Questions

Q: The MoveObstacle script uses += to update transform.position—does that mean it's using vector arithmetic?

A: Yes, and this is a great opportunity to understand vector arithmetic a little better. Input.GetAxis returns a number that's positive if the mouse moves up and negative if the mouse moves down (try adding a `Debug.Log` statement so you can see its value). The obstacle starts at position (–5.75, 1, –1). If the player moves the mouse up and GetAxis returns 0.372, the += operation adds (0, 0.372, 0) to the position. That means it **adds both of the X values** to get a new X value, then does the same for the Y and Z values. So the new Y position is 1 + 0.372 = 1.372, and since we're adding 0 to the X and Z values, only the Y value changes and it moves up.

Make the player navigate around the obstacles



Clear the NavMesh data and re-bake it like you did before: multiselect the Floor, the five Stair, Platform, and Ramp GameObjects in the Hierarchy window, then click the Clear button in the NavMeshSurface component to clear all of the NavMesh data, then select any of those GameObjects and click its Bake button.

There should be holes in the NavMesh on the floor **carved out** around the 8 Ball and Moving Obstacle. Run your game and click on the floor near the 8 Ball—the player will navigate around it.

Now move your player just in front of the moving obstacle. Click on the ramp, and the player takes the long way around, moving all the way to the right, going up the stairs, and back down the ramp.

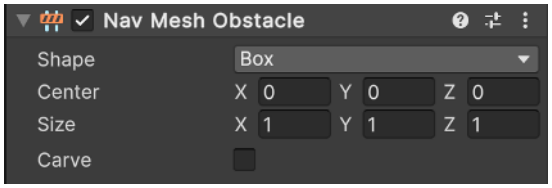
Fix the moving obstacle

While your game is running, move the obstacle up so it's out of the player's way, then click on the ramp again. The player still goes the long way around, even though the obstacle is no longer in the way. What's going on?

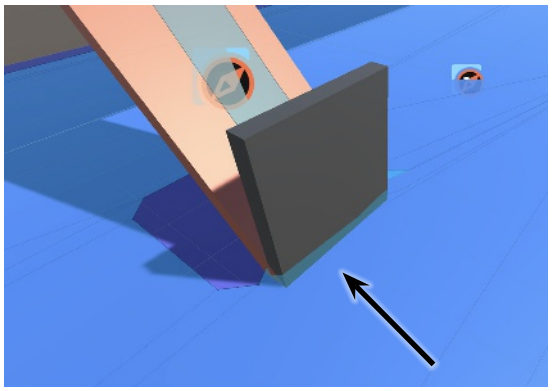
Stop your game, zoom in on the moving obstacle, and take a closer look at the carved-out gap in the NavMesh. The gap is baked into the floor. It doesn't matter where the obstacle is, the gap will still be there.

Luckily, Unity's navigation system has a way to tell the navigation system to handle obstacles that move:

- **Select Moving Obstacle** in the Hierarchy panel.
- Click the Add Component button in the Inspector window, go to the Navigation section, and **add a Nav Mesh Obstacle component**.
- **Clear and re-bake** the NavMesh one more time.



Now that the Moving Obstacle GameObject has a Nav Mesh Obstacle component and the NavMesh has been re-baked, zoom in on it. The carved-out gap around it is no longer there.



Start your game. Position the player in front of the ramp, then click on the ramp. It should start moving toward the ramp, then pause at the moving obstacle. Drag the obstacle up—as soon as there's room for the player to go under it, it will start moving again.

Baking a NavMesh carves out holes for stationary obstacles. Adding a Nav Mesh Obstacle component lets it handle obstacles that move.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas to help you get creative:

- Build out the scene—add more ramps, stairs, platforms, and obstacles. Find creative ways to use materials. Search the web for new texture maps. Make it look interesting!
- Make the NavMesh Agent move faster when the player holds down the Shift key. Search for "KeyCode" in the Scripting Reference to find the left/right Shift key codes.
- You used OnMouseDown, Rotate, RotateAround, and Destroy in the last lab. See if you can use them to create obstacles that rotate or disappear when you click them.
- We don't actually have a game just yet, just a player navigating around a scene. Can you find a way to **turn your program into a timed obstacle course**?

This is your chance to experiment. Using your creativity is a really effective way to quickly build up your coding skills.

You already know enough about Unity to start building interesting games—and that's a great way to get practice so you can keep getting better as a developer.

The Unity Manual has thorough—and readable!—explanations for the various components. Click the Open Reference button (?) at the top of the Nav Mesh Obstacle panel in the Inspector to open up the manual page. Take a minute to read it—it does a great job of explaining the options.

Bullet Points

- When you bake the NavMesh, you can specify a **maximum slope and step height** to let NavMesh Agents navigate ramps and stairs in the scene.
- When a NavMesh Agent moves a GameObject around a scene, it will **avoid obstacles**.
- The label under the Scene Gizmo shows an icon to indicate if it is in **perspective** mode (distant objects look smaller than near objects) or **isometric** mode (all objects appear the same size no matter how far away they are). You can use this icon to toggle between the two views.
- The **transform.LookAt method** makes a GameObject look at a position. You can use it to make the camera point at a GameObject in the scene.
- Calling **Input.GetAxis("Mouse ScrollWheel")** returns a number (usually between –0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move).
- Calling **Input.GetAxis("Mouse Y")** lets you capture mouse movements up and down. You can combine it with OnMouseDrag to move a GameObject with the mouse.
- Baking a NavMesh carves out holes for stationary GameObjects. Adding a **Nav Mesh Obstacle** component to a GameComponent creates an obstacle that carves a hole in the NavMesh that moves as it changes position.
- The Input class has methods to capture input during the Update method, like **Input.GetAxis** for mouse movement and **Input.GetKey** for keyboard input.

Downloadable exercise: animal match boss battle

If you've played a lot of video games (and we're pretty sure that you have!), then you've had to play through a whole lot of boss battles—those fights at the end of a level or section where you face off against an opponent that's bigger and stronger than what you've seen so far. We have one last challenge for you before the end of the book—consider it the *Head First C#* boss battle.

In Chapter 1, you built an animal matching game. It was a great start, but it's missing... something. Can you figure out how to turn your animal matching game into a memory game? Go to our GitHub page and download the PDF for this project—or if you want to play this boss battle in Hard mode, just dive right in and see if you can do it on your own.

There's so much more downloadable material! The book may be over, but you can keep the learning going. We've put together even more downloadable material on important C# topics. We've also continued the Unity learning path with additional Unity Labs and even a Unity boss battle.

We hope you've learned a lot—and even more importantly, we hope your C# learning journey is just beginning. Great developers never stop learning.

Head to our GitHub page for more: *https://github.com/head-first-csharp/fifth-edition*

Our readers mean so much to us. Thank you so much for spending your time with us, and good luck with your C# journey!

☮️❤️🐙 Jenny and Andrew

Thank you for reading our book!

Pat yourself on the back—this is a real accomplishment! We hope this journey has been as rewarding for you as it has been for us, and that you've enjoyed all of the projects and the code that you've written along the way.

But wait, there's more! Your journey's just begun...

In some chapters, we gave you additional projects that you could download from our GitHub page: *https://github.com/head-first-csharp/fifth-edition*

Keep your C# learning journey going by downloading PDFs that continue the *Head First C#* story and cover **essential C# topics**. And while you're there, there's **more to learn about Unity**.

Here are a few things that you can download from our GitHub page:

- The **Head First C# Blazor Learner's Guide**, with Blazor versions of projects in this book to give you a head start with web development in C#.
- A bonus chapter on event handlers and delegates.
- PDF versions of all of the .NET MAUI projects and Unity Labs.
- ***Even more Unity Labs*** that cover physics, collisions, and more, including a complete **Unity Lab project** to create a game from the ground up.

Check out our YouTube channel for walkthroughs and explanations—don't forget to like and subscribe: *https://www.youtube.com/@headfirstcsharp*

Check out these great C# and .NET resources!

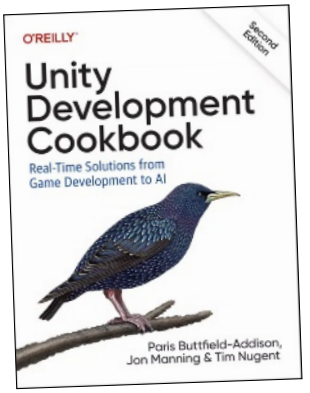
Get connected with the .NET Developer Community: *https://dotnet.microsoft.com/platform/community*

Watch live streams and chat with the team that builds .NET and C#: *https://dotnet.microsoft.com/platform/community/standup*

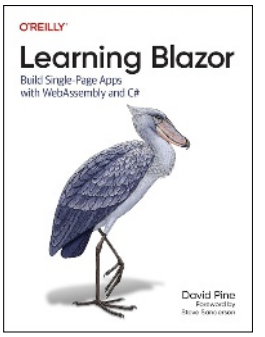
Learn more in the docs: *https://docs.microsoft.com/en-us/dotnet*

Check out these essential (and amazing!) books written by our friends and colleagues, also published by O'REILLY®

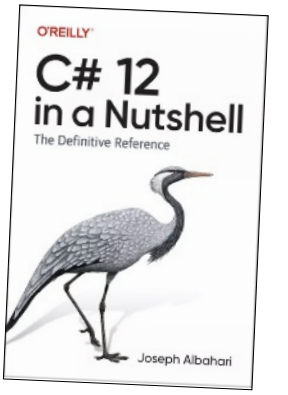
Unity Development Cookbook will help you take your Unity skills to the next level. It's full of incredibly valuable tools and techniques, all in a "recipe" form that you can use right away in your own projects.



Blazor is Microsoft's framework for building web apps in C#. This great book will help you use the C# skills you've learned to become a full-stack web developer.



C# 12 in a Nutshell is a must-have for every C# developer. If you need to do a deep dive into any part of C#, we think you can't do better than this book.



Index

Symbols

++ operator 58, 73

+= operator 217

+ (plus) addition operator 68, 73, 203, 275

< > (angle brackets) 483, 490, 512

< (less than) operator 77

== (equality operator) 77, 78

=> (lambda) operator 563, 598, 609–612

= (assignment) operator 78, 217, 718

*= operator 167, 217

> (greater than) operator 77

|| (OR operator) 77

$ (string interpolation) 276, 457

\0 null character 665

2D view 544

3D environments, Unity scenes as 116

3D vectors 266

&& (AND operator) 77

* (asterisk) multiplication operator 68, 73, 203

\ (backslash) character 456. *See also* escape sequences

@ character
- in arguments 636
- reserving keywords 197
- in strings 456, 639

? character, in reference types 711

// (comments) 66, 70, 171, 219

{ } (curly braces/brackets)
- body of method inside 397
- enums 477
- in if or else blocks 207
- matching up 76
- objects 650
- in string interpolation 505

\\ (double backslash) character sequence 639

- (minus) subtraction operator 68, 203

?? operator 714

() parentheses, method calls 72

; (semicolon), for statement endings 39

/ (slash) division operator 68, 203

[ ] (square brackets) 133, 616. *See also* collection expressions

_ (underscore) pattern 609

/// (XMLDoc comments) 172, 218, 219

A

AAA (Arrange-Act-Assert) pattern 591–592

AbilityScoreCalculator class 211–213

ability score, in RPGs 208–217

AbilityScore project 210–217

abstract classes 394–401, 424–425

abstraction, as principle of OOP 472

abstract methods 394, 397, 438

abstract properties 394, 398, 438

accessibility 94–95, 100, 254–255, 654–655. *See also* screen readers

access modifiers
- and interfaces 433
- internal 596
- protected 452
- public and private 286
- in Visual Studio versus VSCode 139

addition operator (+) 68, 73, 203, 275

Add() method (Dictionary class) 514
Add() method (List class) 489
AI. *See* artificial intelligence
AI-assisted IntelliSense and Intellicode 39, 41
AI Navigation package 678–679
alias 195
allocation 637
Anchor Presets window 545
AND operator (&&) 77
Android device, running .NET Maui on 17, 25
Android SDK 17
angle brackets 483, 490, 512
animal inheritance project 332–338
animal matching game project 18–64
  boss battle 771
  breaking up large projects 20
  C# code for buttons 38–45
  debugging 56–58
  mouse click handling 52–55
  .NET MAUI app project 22–25
  page structure 26
  running the app 41
  timer for 59–63
  XAML code 25–32
anonymous functions 602–603
anonymous types 578, 582
AppendAllText() method 634
Appliance class 443–445
application programming interface (API) 229
AreEqual() method 591–594
AreSame() method 591
args variable 668–670
arguments
  and parameter types 206
  specifying 306
arithmetic operators 203, 205
Arrange-Act-Assert (AAA) pattern 591–592
Array instance 381
ArrayLists 495
arrays 236–237
  converting to lists 495
  creating card deck 481
  creating with interfaces 429
  declaring with var keyword 572
  difficulties with 482
  length 482
  versus lists 484–486, 495
  reference variables in 237
  retrieving objects with [ ] 616
  reusing code in 161
  reversing 660
  string values 133
ArrowDamage class 326–328
artificial intelligence (AI) 187
  adding comments 219
  app packaging/installation 762
  collections 520
  console beeping 671
  discards 762
  generating code with 186–187
  IEquatable<T> interface 716
  interfaces 471
  lambda block bodies 762
  personas 520
  refactoring 310
  separation of concerns and DRY principle 367
  switch expressions 762
  tuples 762
  unit tests 597
as keyword 441, 715
ASP.NET and web development 4
Assert class 591
assignment, of values to variables 71
assignment operator (=) 78, 217, 718
auto-implemented properties 302
automatic properties 433
Awake() method 683

B

backing fields 300, 302

backslash character (\) 456. *See also* escape sequences

Bake feature 679–680, 766

base classes 330–331, 332
- choosing for Zoo Simulator 333–334
- constructors in 359–360
- extending 337–338, 342–343, 376, 378
- subclass access to 358
- upcasting 444

base keyword 358

Beehive Management System project
- class model 375
- converting from turn-based to real-time 390
- DefendHive() method 416–419
- EggCare class 475
- feedback in 388, 389
- IDefender interface 418–419, 426, 429, 448
- interface 370–378
- IWorker interface 430–433, 434–435, 440–441, 448
- NectarCollector class 601–602
- Queen class 381, 464–468

beta programs 586

Billiard Ball game 404–414, 539–552

binary data 220
- encoding 666
- hex dumps 664–667, 669
- hex values for 656
- reading 662
- writing 661

binary (executable) 229

binary numbers 199, 205

binary operators 217

BinaryReader class 662

BinaryWriter class 661

bindings 459

Bird class 510–511

Blazor (Microsoft) 18

block bodies 762

block-scoped namespace 139

blocks of code 69

boilerplate code 719

bool type 71, 192, 200

Border controls, Grid layout 248

boxed objects and structs 704, 720

breakpoints 74–75, 262

bubbling up exceptions 749, 755

buffered data 640

building code 41

Button_Clicked event handler 52–54

Button control 88

buttons and labels, adding in XAML 28–32

byte arrays 633, 639, 660

byte order mark 663, 666

byte type 193, 658

C

C#
- anatomy of a program 128
- benefits of learning 2
- how to learn xxxv–xxxvi
- readability/usability by others 170–185
- scripts 258–259
- specificity of 41
- syntax tips 81
- version used xxxvii
- Visual Studio with 3–11
- VSCode with 12–17

CalculateAbilityScore() method 211

calling a method 69, 72, 131

Canvas 544–546

Captain Amazing 691, 694, 697–698, 715, 719, 727–728, 730

Card class
- adding ToString() method 506
- in CardLinq app 574–575

and CollectionView 526
creating 476–480
prompt engineering challenge 716
reading and writing to files 641–642
record version of 717, 719
sorting with IComparer<Card> 504–505

CardPicker class 132–143
adding to app 136–138
arrays of strings 133, 140–142
finished code 142
namespace style conversion 139
.NET MAUI version 132–138
and Random class 163–164
reusing 154–160

case of text, for public and private fields 299

case statements 489

casting 202–204
calling methods using 417
versus conversion 221
converting random double to other types 245
enums to ints 478
fixing compiler error 212
ints to enums 478
with is keyword 435–437, 440

catch-all exceptions 745–746, 756

catch blocks 742–748

C# Dev Kit 13

Ceiling() method 326

chained LINQ methods 606–607

characters 200
encoding. *See* encoding characters
hexadecimal 656, 664–666
path separator 633
replacement character 666
special, in strings 639
Unicode 195, 653, 655–659, 663

character sheets, in RPGs 191

Character Viewer panel 43

char type 195, 200, 655, 658

Checkbox control, MAUIDamageCalculator 275–278

CheckTemperature() method 708

child classes. *See* subclasses

child controls 248

CIL (Common Intermediate Language) 229

C# Interactive window 188

class diagrams 136, 165, 173

classes 128–185
abstract 394–401, 424–425
access control for instances 288
adding fields to 265
base classes. *See* base classes
building objects from 159–161
collections. *See* collections
concrete 394
and debugger 365
designing 177–179, 366–367
encapsulation 286–291, 293–295, 297
inheritance 330, 375–378
looking for common 335
name best practices 172–173
never instantiated 392–393, 396
private constructors for 322
reusing code with 129
sealed 723
static 166
versus structs 720
subclasses. *See* subclasses

class hierarchy 331, 336, 440, 442

class keyword 128–129, 136–138, 719

class models 339–340, 361–362

clauses 561
from 561
group...by 575–576, 608
join 577
on…equals 577
orderby 561
select 561, 570
where 561

Clear() method 727

Clone class 727–728

Close() method 625–626, 663

Clown class 167–169

CLR. *See* Common Language Runtime

code-behind file 92, 382

code points 656, 659

code snippets 76, 302

CollectionAssert class 594

collection classes 483

collection expressions 133, 253, 496–497, 514

collection initializers 514, 515–516, 519

collections 473, 512–538
- arrays. *See* arrays
- CollectionView 524–538
- copying 519
- dictionaries 512–515, 523, 530, 616
- versus enums 495
- generic 490–495, 516–519
- lists. *See* lists (List<T> class)
- prompt engineering 520
- querying 556
- queues 516–517, 521–522
- versus sequences 617
- stacks 516, 521–522

CollectionView 524–538
- adding Card class 526
- building a two-deck app 533–538
- creating Deck class 527–528
- modifying event handlers 532
- ObservableCollection<T> class 525–531

Collect() method 691

Combine() method 629

ComicAnalyzer class 583–584, 611–612

command-line arguments 668–670, 732

Command Palette, VSCode 17

comments (//) 66, 70, 171, 219

comments (///), XMLDoc 172, 218, 219

Common Intermediate Language (CIL) 229

Common Language Runtime (CLR)
- code execution 229–230, 238
- and constructors 283, 322
- and garbage collection 225
- memory management 229–230, 238, 703
- and object creation 690

Compare() method 500

CompareTo() method 499

comparison operators 77

compiler 135, 188, 197, 201–207, 229

composite formatting 533, 536–537, 633, 639

compound assignment operators 217

concatenating strings 204

Concat() method 557

concrete classes 394

conditional operator 603

conditionals 68

conditional tests, for statements 79

Console App projects 6, 66–67

Console.Clear() method 727

Console.KeyAvailable property 727

Console.OpenStandardInput() method 738

Console.ReadKey().Key method 727

Console.ReadLine() method 131, 242

Console.WriteLine() method 68–69, 283

constants 272, 377

constructors 280, 283
- in base classes 359–360
- exceptions in 749
- initializing properties with 305, 322
- versus inline initialization 313
- multiple 309
- overloaded 475
- with parameters 305
- primary 474–476
- specifying arguments 306

ContainsKey() method 512, 523

Contains() method 492

controls 33, 101–110
Button control 88
Checkbox control 275–278
Entry control 89, 96–99
Grid control 247–250
Image control 88
in mechanics of UIs 88–99
Label control 88, 89
Picker control 89
Stepper control 89, 101
Switch control 89
updating automatically with data binding 459
conversion of types 202–205, 221, 441
Convert class 199, 205, 245
Count() method 560
Count property (Dictionary class) 513
Count property (List class) 489
covariance 510–511
CreateDirectory() method 634
Create() method 661
CryptoStream class 630
curly braces/brackets ({ }). *See* { } (curly braces/brackets)
cylinders 264

D

data binding 459–467, 529–531
data management 189. *See also* types
data objects 459
DateTime.Now method 727
DateTime struct 727
dead objects 693
Debug.DrawRay() method 266
debugger
AbilityScore project error 216
and classes 365
ExceptionExperiment project 744
exception handling 732
exploring yield return 615
finding variable changes 74–75
with loops 76
object referencing itself 235
and overriding 350–351
Paintball Gun class 301
Picker control 106–110
RollDice() method error 280
setting project properties 669
showing objects 505
and time.deltaTime 263
try/catch flow 743–744
Unity game 262
debugging
animal matching game 56–58, 60
C# debug console 13
HexDump project 738, 743
and refactoring 611
unit tests 588, 593
VSCode Command Palette 17
WriteLine() method 281–283
debug logging 758
Debug.WriteLine() method 281, 282, 283
decimal type 194, 200, 202, 220
Deck class
in CardLinq app 610–611
creating 481, 527–528
data binding 529–538
reading and writing to files 641–642
declarative query syntax 561, 564–568, 570–571, 606–607
declaring variables 70
decoding characters 633
default implementations (interfaces) 454, 456
default keyword 327
default values, setting 708
DefendHive() method 416–419
deferred evaluation 573
Dequeue() method 517
dereferencing objects 692
descending keyword 564

deserialization 644
Destroy() method 540
destructors 692–698
diagnostic information, printing 281
dictionaries (Dictionary class) 512–515, 523, 530–532, 616
directories (Directory class) 634–636
disability, and accessibility 654
Dispose() method 637–638, 695–696, 755
DivideByZeroException 735
division operator (/) 68, 203
DLL (dynamic link library) files 759
don't repeat yourself (DRY) design principle 366
double.Parse() method 733
double type 192, 194, 200
do/while loops 79
downcasting 442, 445–447
DrawRay() method 266
DRY (don't repeat yourself) design principle 366
Duck class 497–502, 505, 510–511, 718
DuckComparer class 502
dynamic link library (DLL) files 759

E

edge cases 595
editors, code 67
EggCare class 475
Elephant class 230–235
else keyword 78
embedded system 171
emoji 42–44, 655–656
Empty() method 613
encapsulation 271–324
  for access control 286–291
  best practices 295, 362
  for classes 293–295, 297
  for data safety 298
  for preventing null properties 710
  as principle of OOP 472
  private data with 285
  properties' role in 300
  with protected modifier 452
  versus security 292
  and structs 720
encoding characters 624, 655–672
  BinaryReader class 662
  BinaryWriter class 661
  byte arrays 633, 639, 660
  hex dumps 664–667, 669–670
  text versus binary data 666
  Unicode 653, 655–659, 663
Encoding.UTF8.GetString() method 639, 658
EndOfStream property 629, 665
Enqueue() method 517
Entry control 89, 96–99
entry point 134
Entry_TextChanged() method 99
Enumerable class 613
Enumerable.Range() method 559
Enumerable.Repeat() method 613
enumerable sequences 613–618
enumeration 476–477, 558
Enum.Parse() method 644
enums (enumeration types) 477–481
  casting 478
  versus collections 495
  and lists 501
Environment.GetFolderPath() method 633
Environment.NewLine property 643
equality operator (==) 77, 78
errors 735. *See also* exceptions
  AbilityScore project 216
  filesystem 638
  fixing compiler error with casting 212
  invalid arguments 206
  RollDice() method 280
  syntax 41

escape sequences 639
and strings 195, 456
Unicode in strings 659, 672
event handlers
adding to animal match game buttons 31
Button_Clicked 52–54
editing a method in animal matching game 39
EnteredText control 99
and events 468
modifying to use resource dictionary 532
events 462–463, 468
EventSystem 544
EvilClone class 692, 720, 722
Exception class 737–738
ExceptionExperiment project 744, 760–761
exception filters 754
exception handling 731–761
benefits of 739, 746
catch-all exceptions 745–746, 756–757
catch blocks 742–748, 756–757
choosing the right exception 750
classes and is keyword 437
in constructors 749
custom exceptions 750–752
Dictionary for event handlers 532
dividing by zero 735
exception filters 754
Exception objects 736–739
in finalizers 697
finally blocks 744, 753, 755
handling versus fixing 757
invalid filenames 732
logging 758–760
rethrowing exceptions 749
try/catch blocks 742–748, 754–755
try/finally blocks 755
unhandled exceptions 746, 749
in unit tests 587
unreadable files 740
when to catch/prevent/bubble 755
Exception objects 736–739
exceptions 732–739, 741
DivideByZeroException 396, 735
FileNotFoundException 732
FormatException 734, 737
IndexOutOfRangeException 737
InvalidCastException 734
InvalidOperationException 492
KeyNotFoundException 523
NotImplementedException 61, 750
NullReferenceException 735
OverflowException 734
UnauthorizedAccessException 740
exercises xxxv
AbilityScore project 211, 213–214
abstract classes 399, 400
adding a class, Visual Studio versus VSCode 137–138
arrays 239–240, 485–486
Beehive Management System 374–375, 383–386
Billiard Ball game 411
binary data 661–663
bool-to-string conversion 73
camera 681–682
casting 203–204
Checkbox controls 276–278
class diagrams 273
classes 175
class models 339–340, 361–362
Clown class 167–168
code blocks 345–346
code errors 315–316
code snippets puzzle 311–312
collection expressions 253–254
collections 672
CollectionView 534–538
command line arguments 669–671
conditionals 83
controls 101–102
data binding 464–467, 534–538
debugging 57–58
encapsulation 317–320
enumerable classes 618–619
enums 479–480
exception handling 733–735

extending base classes 347–348
extension methods 725–728
files and directories 635–636, 641–642
filling in Main method 143–144
inheritance 527–528
initial pencil-and-paper 47
interfaces 422–423, 425, 427–428, 431–432, 449–450, 672
is keyword 441, 443
keywords 197–198
lambda expressions 601–602, 610–612
LINQ 559–560, 567–568, 579–580, 583–584, 610–612, 619
lists 485–486, 493–494, 503–505, 507–508
loops 80, 82–86
mouse click event handler 53–54
nullable reference types 711–712
null values 720, 722
object references 227–228, 684, 686
overriding methods 356
PantballGun class 307–308
paper prototypes 149
Picker control debugging 107–108
private keyword 289–290
prompt engineering 187
queues and stacks 521–522
Random class and CardPicker class 163–164
records 721–722
references 231–233
reusing classes 181–184
scene navigation 767–768
serialization 651, 672
streams 627–628, 631–632, 639
structs 701–702
switch expressions 610–612
switch statement 327–328
test-driven development 620
transform.Translate() method 549–551
try/catch blocks 747–748
types 201–202
understanding C# code 48
unit testing 593, 611
using directive 155
value checking 195–196
Visual Studio project 9–10
writing code 363–365
writing to the console 185
XMLDoc 218
exhaustiveness 609
Exists() method 634
ExperimentWithControls project 91–100
adding controls 96–99, 101–110
XAML button details 94–95
explicit types 581
expression-bodied members 600
expressions. *See* collection expressions
extending base classes 337, 342–343, 347–348, 376, 378
extension methods 723–729
Extract Method 581

F

Fail() method 591
feedback loops 388, 389
FeedDog() method 705
field initializer 304
fields
backing fields 300, 302
and classes 165, 265
inline or constructor for initializing 313
and interfaces 452
and objects 165, 285
parameters and masking of 305
private 285–290, 299, 455
public 288, 294, 299
static 454
field values, in Unity 547
FIFO (First In, First Out) 517
File class 634–636, 658, 661, 663
FileInfo class 634
filenames, passing as variables 639
FileNotFoundException 732

file-scoped namespace 139
files, reading and writing 621–672
  appending text 634
  encoding characters 653
  File class 634–636, 658, 661, 663
  finding out if exists 634
  last accessed/modified 634
  serialization 644–652
  streams 622
FileStream class 623–625, 663
finalizers 692–698
finally blocks 744, 753, 755
First In, First Out (FIFO) 517
First() method 560
FlexLayout, XAML 34–37
floating-point types 200
  converting from integers to 206
  decimal 194, 200, 202, 220
  double 192, 194, 200
  errors from 220–221
  float 192, 194, 200, 203
float type 192, 194, 200, 203
flow analysis 710
foreach loops
  versus from clause 570
  using IEnumerable<T> interface 509
  InvalidOperationException 492
  writing objects to console with 506
ForEach() method 721
for loops 79–80
FormatException 734, 737
format strings 633
formatting, composite 633
frame rate 261
frames 261, 262
from clause 561, 570–571
fully qualified class name 505
Func parameter 604, 607
functions 472
FunnyClown class 449–450, 474

G

game design xxxvii, 19. *See also* Unity
  accessibility 654
  aesthetics 314, 387, 389
  dynamics 368, 387, 389
  emergence 368, 391
  mechanics 84–85, 368, 387
  Owen's RPG 190–191, 208–218, 272–282, 361
  prototypes (Ana's game) 145–149
  testing 586
GameObjects 117
  adding behavior to 258–259, 540
  adding material to 120–121, 405
  anchored 545
  attaching tags 548
  Canvas 544
  components 119, 125
  creating 676
  deleting 413
  instances 403–414
  making walkable 683–686, 766
  moving with Move Gizmo 118
  prefab 408
  rendering 117
  transform.Translate() method 549
garbage collection 690–698
  and CLR 238
  consequences of 233
  functions of 229–231
  and null reference 241
  and references 224–225, 429
GC class 691
GC.Collect() method 691, 693, 697
generic collections 490–495, 516–519
generic data types 495

generic lists 490–494
get accessor (getter) 300, 309, 316, 710
GetAxis() method 682, 768
GetComponent<Camera>() method 686
GetFiles() method 634
GetFolderPath() method 633
GetKey() method 681
GetLastAccessTime() method 634
GetLastWriteTime() method 634
GetMouseButtonDown() method 683
GetReviews() method 594–595
GetString() method 639, 658
GetTotalAllocatedBytes() method 691
GetTotalMemory() method 691
Git, saving code with 50–51
Go Fish card game 620
Go To Definition 491
graphs 647–651
greater than operator (>) 77
Grid control 247–250
group...by clause 575–576, 608
Guy class 178–184, 707, 710
Guy object 223–226
GZipStream class 623

H

hacks 434
HashSet 520
heap (memory) 168, 699, 703–704
HeightRequest property 525
hexadecimal characters 656, 664
HexDump project 665–670, 732, 740, 742–743
hex dumps 664–670, 732, 740, 742–743
Hide and Seek game 672
hiding methods 354, 355, 357
hierarchy 331, 337. *See also* class hierarchy
Hierarchy window 260, 411, 413
hit count 262
HorizontalOptions property, controls 95
HUD (head-up display) 148

I

IClown interface 420–421, 449–450, 455, 600
ICollection<T> interface 491
IComparable<T> interface 499
IComparer<T> interface 500–502
ICooksFood interface 446
IDefender interface 418–419, 426, 429, 448
IDE (integrated development environment) 3, 11, 67, 108. *See also* Visual Studio/VSCode
identity functions 608
IDisposable interface 637, 695–696, 753, 755
IEnumerable<T> interface
  building sequences 613–618
  foreach loops using 509
  and sequences 558
  upcasting entire list with 510–511
  and yield return statement 617
IEnumerable interface
  and LINQ 559–560
  Select() method 604
IEnumerator interface 614
IEquatable<string> interface 715
if/else statements 78, 346
if statements 78
Image control 88
immutability 717
Implement interface option 600
implicit types 566, 581
inconsistent accessibility error 584

index 236–237, 486
indexers 513, 616
IndexOf() method 492
IndexOutOfRangeException 737
infinite loops 81
inheritance 325–402
  as principle of OOP 472
  base classes. *See* base classes
  Beehive management interface 370–378
  building class models 331–333, 361–365
  classes advantages 330
  interface 448–450
  multiple 401
  override and virtual keywords 356, 350–357
  paper modeling 361–362
  and structs 699
  subclasses. *See* subclasses
  terminology 332
  Zoo Simulator project 332–336
initialization 180, 305, 313, 322
initializers
  collection 514, 515–516, 519
  field 304
  for statements 79
  object 180, 322
inline assignment 313
INotifyPropertyChanged interface 462–468
Input.GetAxis() method 682, 768
Input.GetKey() method 681
Input.GetMouseButtonDown() method 683
Inspector window 117, 119, 125, 412
instance field 165
instances 160
  access control for 288
  multiple class 158–166
  requirement for, versus static methods 166
Instantiate() method 409, 540
instantiation, class 165
integrated development environment (IDE) 3, 11, 67, 108. *See also* Visual Studio/VSCode
IntelliCode 39
IntelliSense window 39
interface keyword 418
interfaces 418–433, 439, 446–450, 452–471
  versus abstract classes 424–425
  access modifiers 433
  for classes with the same job 439
  data binding 459–467
  default implementations 454, 456–457
  downcasting 446
  inheritance 448–450
  methods and properties 418, 420, 433, 452
  naming 420
  new keyword 426
  and polymorphism 469
  prompt engineering 471
  public 433
  references 426–432, 440–441, 452
  static members 454–455
  upcasting 446
  why use 447, 452, 458
internal classes 590
internal keyword 139, 596
int.TryParse() method 488
int type 71, 192
  assigning decimal value to variable 202
  conversion with + operator 203
  no automatic conversion to string 206
  versus short type 201
intuitive class and method names 172–177
invalid arguments error 206
InvalidCastException 734
InvalidOperationException 492
InvokeRepeating() method 409
iOS devices, running .NET MAUI on 25
IReadOnlyDictionary interface 563

IScaryClown interface 449–450, 457
IsEmpty() method 298–299
IsFalse() method 591
is keyword 435–441, 445, 704, 715
isometric view 765
IsTrue() method 591
IsVisible property 40
iterator, for statements 79
IWorker interface 430–433, 434–435, 440–441, 448

J

JewelThief class 353–355
join clause 577
Join() method 572
JSON serialization 648–652
JsonSerializer 649–652

K

KeyAvailable property 727
KeyNotFoundException 523
keys and values 512–513, 523
Keys property 513
keywords 197–198
kludges 434

L

Label control 89, 98–99
labels and buttons, adding in XAML 28–32
lambda block bodies 762
lambda expressions 598–606, 608, 762
lambda operator 563, 598, 609–613
Language-Integrated Query (LINQ) 553, 555–585, 604–608, 610–612, 619
  calculations on sequences 572
  deferred evaluation 573
  group queries 574–576
  immediate execution 573
  join queries 577, 582
  lambda expressions with 610–612
  method chaining 557, 562, 606–607
  methods 556–560, 604–608
  modifying items 572
  and objects 563–564
  query syntax 561, 564–568, 570–571, 606–607
  and var keyword 580
Last In, First Out (LIFO) 518
Last() method 560
lazy evaluation 573
Length property, arrays 237
less than operator (<) 77
libraries 758–759
LIFO (Last In, First Out) 518
line breaks 456
line endings 643
LINQ. *See* Language-Integrated Query
Linux
  .NET MAUI and Android device 17
  .NET MAUI installation 17
  and Visual Studio/VSCode 3, 12
lists (List<T> class) 42, 483–511
  accessing element at specific index 486
  versus arrays 484–486, 495
  converting to arrays 495
  creating 483, 495
  creating with collection expressions 496–497
  generic 490–492
  iterating through 492, 721
  sorting 498–504
  using [ ] to return object from 616
  writing to console 505–509
lists, nongeneric 495
literals 196, 221
Locksmith class 352
Log class 760

logging 758–760
logging libraries 759
logical operators 77
long type 193
LookAt() method 770
loops 68, 79–80
  code snippets for writing 76
  foreach. *See* foreach loops
  infinite 81
  nested 583
Lumberjack class 521–522

M

macOS
  accessibility 95
  Character Viewer panel 43
  line endings 643
  .NET MAUI installation 17
  and Visual Studio/VSCode 3, 4, 12
  VoiceOver 254
MAGAZINE_SIZE constant 304
Main Camera 675, 686
Main method 134–135, 143–144
ManualSportSequence class 614–616
Math.Ceiling() method 326
MAUIDamageCalculator app 273–281
Max() method 560
Mechanics-Dynamics-Aesthetics (MDA) framework 389
mechanics of software design 84–85, 368, 387. *See also* controls
memory
  allocating 201
  heap 168, 699, 703–704
  stack versus heap 703–705
  and types 200
MemoryStream class 623, 639–640
Message property 737
metacognition xxxiii
method chaining 557, 562
methods 39
  abstract 394, 397
  accessing private fields with public methods 288
  accessing with is keyword 436–439
  and variables. *See* variables
  and anonymous types 582
  body 394, 397, 600
  calling most specific 337
  calling risky 741
  defining with interfaces 418
  extension 723–729
  extracting 581
  versus get and set accessors 309
  hiding 354, 355, 357
  name best practices 172–173
  object 159
  operators in 73
  overloaded 628
  override and virtual keywords 356
  overriding 334, 344, 350–351, 505
  passing arguments by reference 707
  public 294
  returning multiple values 706
  setting default values 708
  signature 322
  and statements 69–75
  static 166, 454
  syntax for calling 72
  this keyword with 234
  virtual 355
Microsoft Visual Studio Debug Console window 8
Min() method 560
mod function 221
ModifyAnIntAndGuy() method 707
Moods class 460–463
mouse click handling 52–55
Move Gizmo 118
MoveNext() method 509

MoveObstacle script 767–769
MoveToClick() method 764, 767
MSTest 587, 591, 594
multiple inheritance 401
multiplication operator (*) 68, 73, 203
myStrings variable 140

N

named arguments 708
names, for variables 70
namespaces 139, 132–143, 155, 530
navigation and pathfinding system 673–686, 763–770
navigation system 677
NavMesh 677–678, 764, 767, 769
NavMesh Agent 679, 686, 764
Nav Mesh Obstacle component 769, 770
NectarCollector class 601–602
nested loops 583
nested .NET MAUI controls 33
nested tags 97
.NET Core SDK 15
.NET desktop development tools 4
.NET MAUI app 18, 22–25. *See also* XAML
  as game design tool 21
  controls 88–99
  creating in Visual Studio 22
  creating in VSCode 23
  installing in VSCode 13, 17
  running 24
.NET version 7
NetworkStream class 623
new keyword
  array of strings 140
  creating anonymous types 578
  creating object from class 160
  instantiating a class 158, 592
  interfaces 426
new statements 159, 236
NextDouble() method 245
\n (line feed character) 195, 456
nongeneric collections 509, 523
non-nullable reference types 710
non-numeric types 200
NotImplementedException 61, 750
NRE (NullReferenceException) 709–710, 735
NuGet package manager 758–759
nullable reference types 711–712
nullable value types (Nullable<T> struct) 713
null-coalescing operator (??) 714
null keyword 241–242, 276
NullReferenceException (NRE) 709–710, 735
null references 709–713
numbers, representing with names using enums 478

O

object initializers 180, 322
object keyword 704
object-oriented programming (OOP) 472
object references 452, 684, 686, 690
objects
  accessing fields inside 285
  boxed 704
  building from classes 159–161
  checking type 435
  dereferencing 692
  Dispose() method 637–638
  encapsulation. *See* encapsulation
  and fields 165
  garbage collection 224–225, 690–698
  inheritance. *See* inheritance
  and LINQ 563–564
  misusing 284

for multiple class instances 158–166
null keyword 241–242
object type 195
references 429
serialization 644–652
storage in heap memory 168
versus structs 699, 701–702, 704, 715
talking to other objects 234–235, 238
ToString() method 505–509
versus value types 700–701

Object.ToString() method 142, 505

ObservableCollection<T> class 525–531

OneBallBehaviour class 540–543, 549–551

on…equals clause 577

OnMouseDown() method 540

OnMouseDrag() method 767

OOP (object-oriented programming) 472

opaque box, object as 293

opaque box tests 596

OpenStandardInput() method 738

OpenWrite() method 661

operating systems, building code to run on 55
(see also specific systems by name)

OperatorExamples() method 72–73

operators 68, 73, 77

optional parameters, setting default values with 708

orderby clause 561, 564

OrderByDescending() method 608

OrdinaryHuman class 723

OR operator (| |) 77

out modifier 706

out parameters 706

Output window 281, 282

OverflowException 734

overloaded constructors 475

overloaded methods 479, 628

override keyword 344, 351, 352–353

overriding methods 344, 505
debugging 350–351
versus hiding 354, 357
override keyword 356

P

packages 758–759

pages, .NET MAUI 33

PaintballGun class 298–304, 475

paper prototypes 147–149

parameterless constructor 305

parameters 130–131
versus arguments 206
constructor 305
masking fields 305
and set accessors 309

parentheses (), method calls 72

parenting 676

parent, in inheritance 332

passing by reference 707

passing by value 707

Path.Combine() method 629

Path.DirectorySeparatorChar field 633

pathfinding and navigation system 673–686, 763–770

path separator characters 633

patterns 591

personas, asking AI to adopt 471, 520

perspective view 765

physics engine 413

Physics.Raycast() method 685, 764

pickedCards variable 140

Picker control 89

PickRandomCardsMAUI project 151–152

PickRandomCards project 132–143. *See also* CardPicker class

PickSomeCards() method 133, 140–141
placeholder objects 674
Placeholder property 97
placeholder text, Entry control 97
Plane object 674
platforms, in scenes 765–766
playtesting 586
polymorphism 469
Pop() method 518
prefab 408, 540–541, 548, 680–681
primary constructors 474–476
primitive objects 674
private fields 285–290, 299
private keyword 291–295
private setter 303
Program class, PickRandomCards project 134
Program.cs 67
project management 6. *See also* animal matching game project
project references 589
project templates 6
Project window 259
prompt engineering 187
  adding comments 219
  app packaging/installation 762
  collections 520
  console beeping 671
  discards 762
  exploring C# with 86–87
  generating code with 186–187
  IEquatable<T> interface 716
  interfaces 471
  lambda block bodies 762
  refactoring 310
  separation of concerns and DRY principle 367
  switch expressions 762
  tuples 762
  unit tests 597
  value equality 716
properties
  abstract 394, 398
  auto-implemented 302
  defining with interfaces 418
  encapsulation role of 300–303
  Grid control 248
  initializing with constructors 305
  inline or constructor for initializing 313
  in interfaces 433, 452
  statements in 309
  static 454
Properties window, Visual Studio 99
property accessors 300
PropertyChanged event 463
protected keyword 349, 452, 455
prototypes 145–149
public classes 590
public fields 294, 299
public interfaces 433
public keyword 139
public methods 288, 294
Push() method 518

Q

quality assurance (QA) tests 586
queries. *See* Language-Integrated Query (LINQ)
queues 516–517, 519
quick actions, removing using directives 138
quotation marks, escaping in strings 639

R

Random class (C#) 206, 245–254, 479
Random class (Unity) 407
Random.value property 407

random number generators 244
RandomSuit() method 141
RandomValue() method 141–142
Range() method 559–560
Raspberry PI 170
raycasting 684, 764
Raycast() method 685
ReadAllBytes() method 658, 663
ReadAllLines() method 663
ReadBlock() method 665
ReadKey().Key method 727
ReadLine() method 131
Read() method 623, 666, 667
readonly keyword 451, 719
read-only property 303
real-time game 390
records (record class) 717–722
record struct 719
recursion 705
refactoring 174
  AI assistance with 310
  with lambda expressions 598, 600
  steps in 300
  unit tests 611
refactoring tools 581, 605
references and reference variables 222–228, 238
  in arrays 237
  copying references 223
  equals sign with 700–701
  garbage collection 224–225
  interface references 426–432, 440–441, 452
  multiple references to single object 226–228, 233, 437, 446
  nullable and non-nullable reference types 710
  null references 709–713
  object references 452, 684, 686, 690
  objects talking to other objects 234–235
  passing by reference 707
reference types, nullable and non-nullable 710
ref keyword 707
relational operators 77
remainders 221
RemoveAt() method 489
Remove() method 513
Rename feature 575
Repeat() method 613
replacement character 666
reserved keywords 197–198
Reset Window Layout command 11
resource dictionaries 530–532
resource management game 387
return statement 69, 130, 131, 141
return type 130, 316
return values 130–131
reusing code with classes 129
Reverse() method (Array class) 660
Reverse() method (LINQ) 560
risky code 741–758
RoboBee class 430
robust design 595, 740
role-playing games (RPGs) 190–191
RollDice() method 280
root, of graphs 649
Rotate Tool 123
rounding in interpolation 347
rows and columns, Grid layout 249
RPGs (role-playing games) 190–191
runtime 741
runtime errors. *See* exceptions

S

Safe class 352–355

sbyte type 193

ScareAdults() method 457

ScaryClown class 449–450, 601–602

Scene Gizmo 124

Scene view 114, 267, 269

ScreenPointToRay() method 685, 764

screen readers 94, 100
- announcing text 321
- PickRandomCardsMAUI project 150
- Sloppy Joe app 254–256
- and Unicode characters 655

Script components 547

ScrollView control 91

sealed classes 723

SecretAgent class 286–288

security, versus encapsulation 292

Seek() method 623

select clause 561, 564, 570, 604–605

SelectionChanged event handler method 526

SelectionMode property 526

Select() method 604

self-closing tags (XAML) 33

SemanticProperties, controls 94

SemanticProperties.Description property 255

SemanticProperties.Hint property 97

semicolon (;), for statement endings 39

Sens-AI 86–87. *See also* prompt engineering

separation of concerns 366–367, 434

sequences
- versus collections 617
- and LINQ 558–561, 572

serialization 644–652, 672
- and finalizers 695
- with JSON 648–652
- of object graphs 647

Serilog logging library 759–760

set accessor (setter) 300, 303, 309, 316, 710

SetDestination() method 686

SetValue() method 256

Shoe class 487

ShoeCloset class 488–489, 491

short type 193, 201

Shuffle() method 575

signature (method) 322

signed types 193

significant digits 194

Skip() method 560

Sleep() method 727

Slider control 89

Sloppy Joe's Random Menu Item project 246–254

Solution Explorer
- group query process 574
- logging library 759
- PickRandomCards project 136–137
- TwoDecks exercise 534
- unit tests 587–589
- Visual Studio and VSCode 10, 16–17, 23

solutions, in Visual Studio 596

Sort() method 498–499

Split() method 641

square brackets ([ ]) 133, 616. *See also* collection expressions

stack (memory) 703–705, 720

stacks (Stack class) 516, 518–519

StackTrace property 737

stack traces 593

Standard Error (stderr) 671

StartGame() method 546, 548

statements 39, 66
- loops 79–80

and methods 69–75
in properties 309
states, saving 645–646
static fields 454
static keyword 166
static methods 166, 454–455
static properties 454
stderr (Standard Error) 671
Stepper control 89, 101
Stopwatch class 692
StreamReader class 629, 631–633, 663, 665–667
streams (Stream class) 622–644
chaining streams 630
closing 633, 637–638, 640
Directory class 634–636
File class 634–636
FileStream class 623–625, 663
flushing buffered data 640
hex dumps with 664–667
MemoryStream class 639–640
StreamReader class 629, 631–633, 663, 665–667
StreamWriter class 625–629, 631–633, 640–643, 663
types of 623
using statements 638
StreamWriter class 625–629, 631–633, 640–643, 663
String constructor 665
String.Empty() method 714
string interpolation 275–276
string literals 456
strings 195, 655
@ character in 456, 636, 639
backslashes in filenames 639
concatenating 204, 205, 572
constructor 665
converting to byte array 633
ends represented by \0 665
extending 724
formatting for printing 633
line breaks in 456
returning empty 714
special characters 639
System.String class 715
string type 71, 192, 200, 205, 243
string? type 243
string values 131, 133
structs 699
boxed 704
versus classes 720
in games 705
versus objects 701–702, 704, 715
record structs 719
setting one equal to another 701, 720
subclasses 330
accessing methods with is keyword 436–439
and base classes 337, 338, 343–345, 358
constructors in 359–360
implementing members in 349
modifying 343–344
overriding methods 344
upcasting 444
subtraction operator (-) 68, 203
Sum() method 560
superclass 332, 338, 341, 355
Switch control 89
switch expressions 609–613
switch statements 327, 346, 489, 609
SwordDamage class 272–279, 297
syntactic sugar 753, 755
syntax errors in code 41
System.Exception 745, 750. *See also* exceptions
System.Object 195
System.String class 195, 715

T

tag-based markup language, XAML as 28
tags 49, 548
TakeLast() method 557

Take() method 557
T? alias for Nullable<T> 713
TallGuy class 420, 600
target-typed new expressions 592
ternary operator 603
test-driven development 620
Test Explorer window 588
testing
  test-driven development 620
  unit tests 587–597, 611
  ways to test 586
text data, encoding 666
Text property, controls 94
this keyword 234–235, 238
  calling primary constructors 475
  in extension methods 723
  initializing properties with constructors 305
Thread.Sleep() method 727
throw statements 749
time.deltaTime 261, 263
timers, adding to animal matching game 59–63
TimerTick() method 60–63
<T> notation 490
ToArray() method 639
ToList() method 721
Toolbox window, Visual Studio 96
top-level statements 135
ToString() method (Convert class) 205
ToString() method (Object class) 142, 505–506
Transform component 119, 676
transform.LookAt() method 770
transform.Rotate() method 261
transform.Translate() method 549–551
try/catch blocks 742–748, 754
try/finally blocks 755
TryGetValue() method 523
TryParse() method 707
\t (tab character) 195
tuples 762
turn-based game 390
type arguments 490
type conversion 202–205, 221, 441
types 192–207
  arrays 236–237
  automatic casting in C# 205
  char 195, 200
  checking with is keyword 435
  declaring implicitly typed arrays 572
  enums 478, 477–481, 495, 501
  generic 495
  implicit/explicit 566, 581
  in JSON data 651
  of literals 196
  and memory 200
  non-numeric 200
  object 195, 222–224
  reference. *See* reference types
  return type 130
  string 243
  value. *See* value types
  variable 70–71, 199
  and var keyword 566–570

U

\u escape sequence 659, 672
\U escape sequence 672
uint type 193
UI (user interface) 539–552
ulong type 193
UnauthorizedAccessException 740
unboxing objects 704
underscore (_) pattern 609
unhandled exceptions 746, 749
Unicode 195–196, 653, 655–659, 663

unit tests 587–597, 611
Unity 111–126
building games in 404–414
creating projects 114
extension installation 13
GameObjects. *See* GameObjects
installing with Unity Hub 113–114
layout controls 115
pathfinding and navigation system 673–686, 763–770
raycasting lab 673–686
scene navigation lab 763–770
user interfaces lab 539–552
versions 113
Unity Hub 113
Unity Version Control 122
unsigned types 193
upcasting 442–444, 446–447, 510–511
Update() method 261
user interface (UI) 539–552
ushort type 193
using directives 138, 155–156
using statements 638, 753, 755
UTF-8 656, 658–659, 663
UTF-16 656, 659
UTF-32 659

V

value equality 716–717
value parameter, set accessors 309
values
in arrays 236
checking 195
of variables 70–71
wrap around in casting 203
value types
bool 71, 192, 200
byte 193
changing 238
converting 707
decimal 194, 200, 202, 220
double 192, 194, 200
float 192, 194, 200, 203
floating-point 194, 200, 206, 220–221
int. *See* int type
long type 193
nullable 713
versus objects 700–701
sbyte 193
short 193, 201
signed 193
string 71, 192, 200, 205
structs as 701
uint 193
ulong 193
unsigned 193
ushort 193
variables 68, 70–72, 199
assigning values to 202
debugger to find changes 74
declaring 70, 566, 569
and generating methods 72
operators with 77
values of 70
watches for values 74–75
var keyword 566–570, 572, 580
vector arithmetic 768
verbatim string literals 456, 457
VerticalStackLayout controls 91
video game design. *See* game design
View Tool 124
virtual keyword 344, 351, 352–353, 356
virtual methods 355, 456
Visual Studio Community Edition 4
Visual Studio/VSCode 12–17
access modifiers 139
adding new class 136
C# Interactive window 188
command line arguments 669

creating and running a project 3–11, 14–17
default interface implementations 454
editions and versions of 4, 11
event handler code 31
helping you code 67
hit count for breakpoints 262
installing 4
installing C# extensions 13
JSON 650
lack of Locals window in VSCode 75
lack of Toolbox in VSCode 97
Program.cs 67
refactoring tools 581, 605
Rename feature 575
saving code with Git 50–51
solutions in 596
testing 587–593
Toolbox window 96
and Unicode 657
Unity setup 115
version used xxxvii
XAML support limitations 27, 28
XML documentation help 172

void return type 173

VSCode. *See* Visual Studio/VSCode

W

watches, for variable values 74–75

where clause 561, 607

while loops 79

Windows
accessibility 95
emoji panel 43
line endings 643
Narrator app 254
and Visual Studio 3, 12

with keyword 718

WriteAllBytes() method 663

WriteAllLines() method 663

WriteAllText() method 659

WriteCards() method 641–642

WriteLine() method (Console class) 68, 69, 283

WriteLine() method (Debug class) 281, 282, 283

WriteLine() method (StreamWriter class) 625–626

Write() method (BinaryWriter class) 661

Write() method (StreamWriter class) 625

X

XAML 25–32
button details 94–95
buttons and labels 28–32
data binding 529–531
editing code with 27
ExperimentWithControls project 91
FlexLayout 34–37
Grid layout 251

Xcode for macOS 17, 24

XML Documentation Comments (XMLDoc) 172, 218, 219

Y

yield return statement 615–617

Z

zero-based arrays 236

Zoo Simulator project 332–336, 337

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2023 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 175.8 x 9.25